Assignment 3 REQ 1: Limveld

This design rationale covers the two main features for the implementation of the Limveld world:

1. Travelling to Limveld: Implementing teleportation between game maps
2. Creatures of Limveld: Enhancing behaviour selection for NPCs with different decision-making strategies

Design Implementations and Evaluations:

Implementation A: Coding the Limveld GameMap and adding it as a teleport location within the Application class

Pros	Cons
Allows for clean separation of map data from game logic	Teleport locations hardcoded (valleyTeleportLocation at 22,13) require code changes to adjust positions
Easy to modify the map layout by merely changing the string array	You cannot dynamically generate or modify maps during runtime
Simple to add any additional, connected maps in the future promoting extensibility	

Example Scenario: Suppose a level designer wants to reposition the teleport location due to new terrain blocking the original spot

- **Maintainability:** As the teleport location is hardcoded in the Application class, even a small change like this requires modifying the core game logic.
- **Improvement for Future Implementation:** A better design can be considered by allowing teleport destinations to be configurable

Implementation B: Create a specialised LimveldPortal ground type that enables teleportation when interacted with

Pros	Cons
Encapsulates all teleportation logic for this portal in one class	Portal destinations are fixed after creation
Only actors with IS_PLAYER status can activate (clean permission system)	Requires manual setup/hardcoding in the Application class
TeleportAction handles movement separately from portal detection	Cannot represent disabled or one-way portals without modification

Example Scenario: The game expands with a new kind of portal that only works during night-time

- **Maintainability:** As the Limveld Portal encapsulates its core logic, it's possible to create a subclass called NightPortal and override factors such as activation conditions without touching other parts of the codebase
- **Benefits:** This supports the Single Responsibility Principle and allows for polymorphic behaviour, enhancing flexibility

Implementation C: Created a BehaviourSelection interface to abstract behaviour selection strategies

Pros	Cons
Single interface enforces simplicity	Interface doesn't enforce behaviour priority systems
Explicitly handles null returns for no-action cases	

Example Scenario: You want to introduce a new AggressiveBehaviourSelection method that prioritises combat related behaviours first for NPCs

- **Maintainability:** Using the BehaviourSelection interface you can implement a new concrete class and inject it into NPCs without modifying existing classes (via the Application class)
- **Benefits:** This design respects the Open/Closed Principle and Dependency Inversion Principle, because the behaviour of NPCs rely on abstraction rather than concrete classes

Implementation D: Created a concrete PrioritisedBehaviourSelection class which implements the BehaviourSelection interface

Pros	Cons
Always processes behaviours in priority value order (1,2,3...)	Cannot dynamically reprioritise behaviours
Returns immediately on first valid action (efficient for busy actors)	

Implementation E: Created a concrete RandomBehaviourSelection class which implements the BehaviourSelection interface

Pros	Cons
All behaviours have identical selection probability	Converts to array every turn, which can be memory intensive
Explicit empty map check prevents exceptions	Doesn't verify behaviours through array contents

Implementation F: Remodifying constructors in the PeacefulCharacter abstract class so it can accommodate behaviour selection strategies (prioritised by default but configurable if needed)

Pros	Cons
Has an optional BehaviourSelection parameter with default	Hardcodes PrioritisedBehaviourSelection
Retains original constructor signatures	Directly depends on Behaviour implementations, which means there is tight coupling

Example Scenario: You want to test how NPCs behave with RandomBehaviourSelection in simulations for debugging purposes

- **Maintainability:** Because the constructor accepts a BehaviourSelection strategy, testing different NPC behaviour strategies becomes easy and isolated, thereby reducing the need for subclasses and overengineering
- **Benefits:** This improves testability and adheres to Dependency Injection, making unit tests cleaner and promoting loose coupling.

Alternative Design Comparisons:

1. ***Hardcoding Behaviour strategies in each NPC sub-classes playturn methods***

Pros	Cons
Extremely simple to implement since there is no need to implement dedicated interfaces	Every NPC class would need its own copy of selection logic, duplicating code
Provides direct control since all behaviour logic is contained within one class	Changing selection logic requires editing every NPC class
	Hard to test selection logic independently of NPC class

Why it was Rejected:

- Adding more behaviour selection strategies would mean we would need to:
    - Modify all NPC classes
    - Break existing test cases for each class
- Violates the Open/Closed principle by requiring modifications to existing classes for new features
- Makes the code base more complex and larger (N strategies × M NPC classes) due to duplication

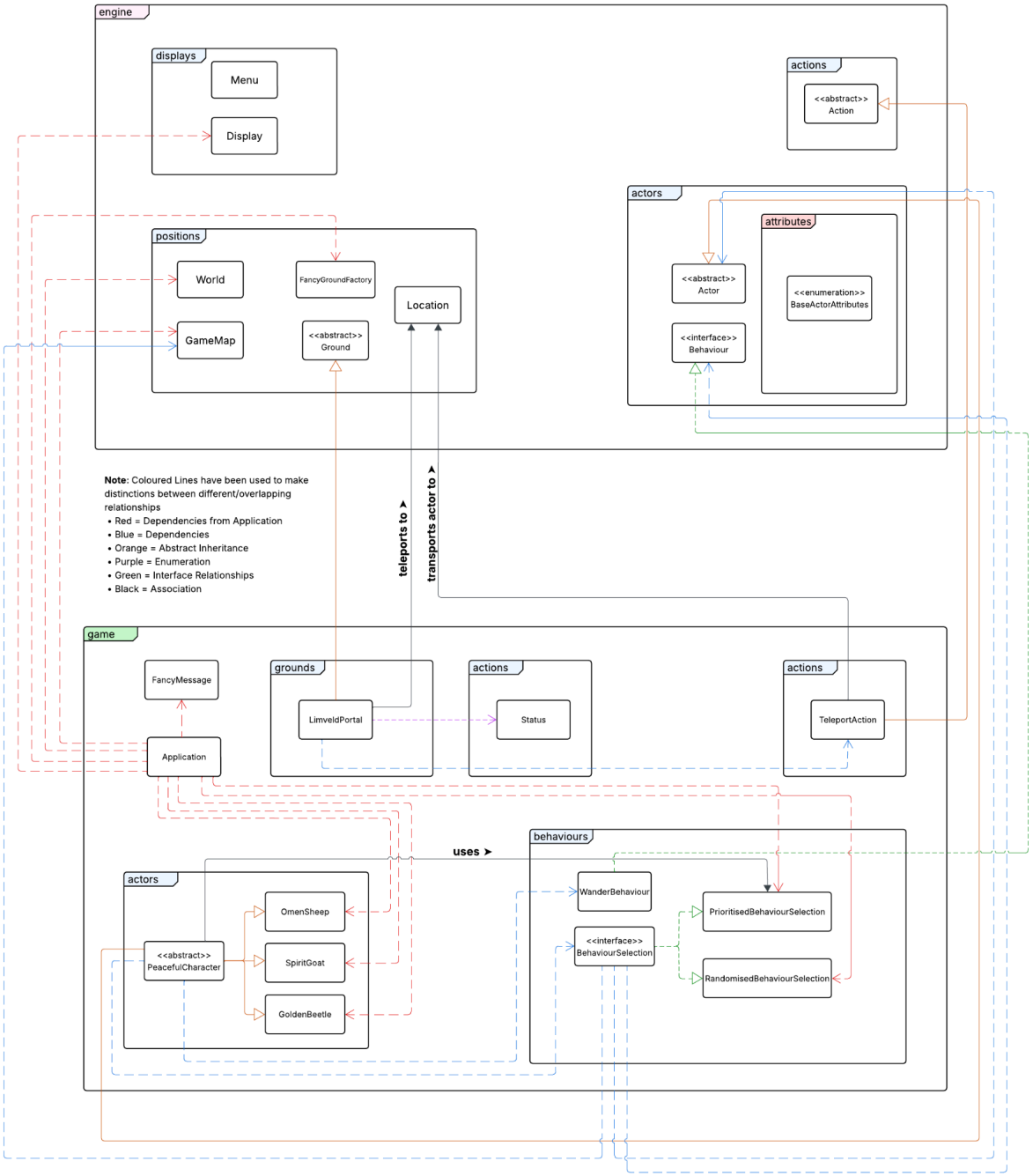
2. ***Behaviour strategies are chosen by enumerations within NPC classes***

Pros	Cons
Simpler to implement than interface abstraction as it only requires an enumeration and/or switch logic	Still places selection logic inside the NPC class, violating Separation of Concerns
Avoids creating multiple small classes for each strategy	Makes testing individual strategy logic harder unless refactored out into helper methods
	Violates the Open/Closed Principle since it involves adding new strategy types requires changes to every NPC using the enumeration

Why it was Rejected:

- Although using enumerations simplify the code over hardcoding each strategy fully in NPC classes, they still couple behaviour selections to the each NPC class, making strategy management rigid and non-extensible.
- It still fails to support polymorphism and forces behaviour logic to remain inside each NPC class, making it harder to test and reuse.

UML Diagram Analysis:



The diagram shows the implementation of the Limveld requirements through object-oriented design. The LimveldPortal class extends the ground parent class with a clear association relationship with the Location class, encapsulating teleportation mechanics within a single-focussed component. This design adheres to the Single Responsibility Principle as it isolates all portal related functionality for the Limveld Portal within one class, whilst maintaining a clear connection to its paired destination. It also ensures that the portal can also be assigned different teleportation locations upon initialisation (by modifying its teleportation attribute in the Application class).

The teleportation system demonstrates strong design principles through its clear separation of concerns. The LimveldPortal class is responsible for managing the visual and interactive representation of teleportation points, while the GameMap class oversees the location management of different game maps upon initialisation. This logic is therefore brought together by the TeleportAction class which encapsulates the transition logic for actors between maps. TeleportAction ensures that only actors with the IS_PLAYER status are permitted to teleport, preserving game logic integrity by enforcing capability-based access control. By delegating the movement logic to the GameMap class and maintaining its own responsibility solely for validating eligibility and triggering the transfer, the TeleportAction class demonstrates the Single Responsibility Principle.

The design also showcases the application of unique behaviour strategies through the BehaviourSelection interface and its implementations, RandomBehaviourSelection and PrioritisedBehaviourSelection. These classes provide flexible alternatives for NPC decision making. This adheres to the Interface Segregation Principle by ensuring that NPCs only depend on the behaviour selection logic they actually use, without being forced to implement irrelevant methods. It also supports the Open/Closed Principle, as new selection strategies can be introduced without altering existing NPC classes.

NPC characters such as the SpiritGoat and OmenSheep classes demonstrate inheritance design by extending the PeacefulCharacter class. The clever use of dependency injection for behaviour selection strategies demonstrates adherence to the Dependency Inversion Principle, as high-level NPC classes depend only on the BehaviourSelection abstraction. The protected access to behaviours in PeacefulCharacter provide ideal flexibility for subclasses while maintaining the Liskov Substitution Principle.

The design's modular structure ensures strong future extensibility. New behaviour strategies can be added via additional BehaviourSelection implementations, and portal functionality can be extended edited directly through the Application Class to modify teleportation destinations for the LimveldPortal. The PeacefulCharacter hierarchy supports the easy integration of new NPC types, while the GameMap system allows seamless addition of new environments. This architecture promotes scalability without compromising maintainability.