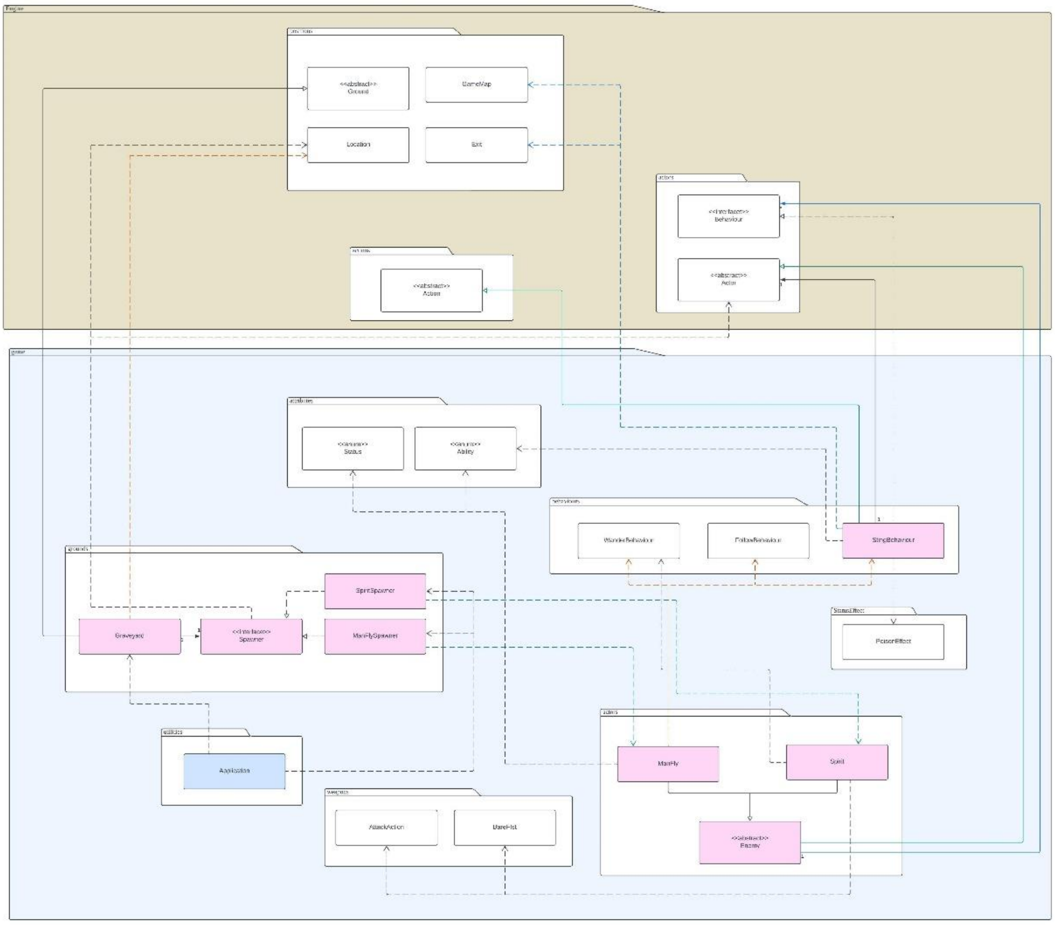
REQ3: Ailment



Pink = Class Created

Blue = Class Modified

REQ 3 UML Diagram

<table>
<tr><th>Classes Modified / Created</th><th>Roles and Responsibilities</th><th>Rationale</th></tr>
<tr>
<td>Created enemy abstract class (extends Actor)
and
Use tree map for behaviours</td>
<td><b>Role &amp; Responsibility:</b>
Serves as the base class for all enemy types, sharing common attributes and behaviors.
Implements the playTurn() method to iterate over behaviors stored in a TreeMap<Integer, Behaviour>, executing the appropriate action based on the game state.
<b>Relationship:</b>
- Extends Actor.
- Has a Map<Integer, Behaviour> to store behaviors with priorities.
- ManFly and Spirit are subclasses of Enemy.
-</td>
<td rowspan="2"><b>Alternative Solution: Concrete enemy class</b>
Make Enemy a Concrete Class Without Behaviours and Move All Logic into ManFly and Spirit.
<b>Pros and Cons</b>
<table>
<tr><th>Pros</th><th>Cons</th></tr>
<tr><td>Simpler structure with fewer behavior classes.</td><td>Violates the Open/Closed Principle (OCP): Adding new enemy types (e.g., Skeleton, Zombie) would require modifying the Enemy class to add specific logic.</td></tr>
<tr><td>Reduced Complexity: No need for a TreeMap to manage behaviors.</td><td>Code Duplication: Each enemy class would need its own stinging, wandering, or following logic, leading to duplication..</td></tr>
</table>
<b>Finalised Solution:</b>
I created an abstract <b>Enemy</b> class, which extends <b>Actor</b> and uses a <b>TreeMap</b> to manage enemy behaviors. This class provides a common structure for all enemies, with behaviors prioritized and executed based on game conditions. From this base, I implemented specific enemies like <b>ManFly</b> and <b>Spirit</b>. <b>ManFly</b> has a <b>StingBehaviour</b> for attacking and poisoning the player, along with a <b>FollowBehaviour</b> to track the player across the map. On the other hand, <b>Spirit</b> only wanders using the <b>WanderBehaviour</b> and attacks nearby players with a basic <b>BareFist</b> attack.
Additionally, I created the <b>StingBehaviour</b> class, which handles stinging attacks that have a chance to poison the player. This behavior acts both as an action and a behavior, keeping the logic for attacking modular and flexible for other enemy types.</td>
</tr>
<tr>
<td>Created ManFly class
(extends Enemy)</td>
<td><b>Role &amp; Responsibility:</b>
ManFly is a type of enemy that sting and poison the player.
uses the StingBehaviour to attack the player with a chance to apply poison.
It can follow the player across the map and has a specific resistance to poison.
<b>Relationship:</b>
- Extends Enemy.
- Uses StingBehaviour to perform sting attacks.
- Uses FollowBehaviour to follow the player.
- ManFly Spawner spawns ManFlies.</td>
</tr>
</table>

Created Spirit Class (extends Enemy)	**Role & Responsibility:** Spirit is a type of enemy that can wander and attack players within range. It differs from ManFly by not following the player (it only attacks when in range). **Relationship:** - Extends Enemy. - Uses Wander Behaviour to move around. - Uses BareFist to attack the player. - Spirit Spawner spawns Spirits.	**Reasons for Decision:** 1. Single Responsibility Principle (SRP): The Enemy class handles enemy logic, while specific behaviors are managed by separate classes. 2. Open/Closed Principle (OCP): We can add new enemies or behaviors without changing existing code. 3. Liskov Substitution Principle (LSP): Any enemy subclass (like ManFly or Spirit) can be used in the game without breaking the logic. 4. Dependency Inversion Principle (DIP): The system relies on behavior abstractions rather than concrete implementations, allowing greater flexibility. 5. DRY Principle: Shared behavior logic prevents code duplication across different enemies. **Limitations & Trade-offs:** 1. Increased Complexity: Managing behavior priorities with a TreeMap adds some complexity in deciding which behaviors take priority. 2. Performance Overhead: Iterating over behaviors for every turn might slightly affect performance, but this is only noticeable with a large number of enemies. 3. Memory Usage: Each enemy stores a map of behaviors, increasing memory usage, but the added flexibility and maintainability are worth it.
Created StingBehaviour class (extends Action implements Behaviour)	**Role & Responsibility:** Implements a stinging attack with a chance to poison the player. Acts as both an Action and a Behaviour that can be executed by enemies like ManFly. **Relationship:** - Implements Action and Behaviour. - Used by ManFly to sting the player and apply poison effect if successful.	

<table>
<tr>
<td>Created Spawner Interface</td>
<td><b>Role &amp; Responsibility:</b>

Defines a contract for spawning new entities (e.g., enemies).

<b>Relationship:</b>

Implemented by both ManFly Spawner and Spirit Spawner to handle enemy spawning logic.</td>
<td rowspan="3"><b>Alternative Solution:</b>
Instead of using an interface, the spawning logic could be combined into one Spawner class using instanceof to handle different types like ManFly and Spirit.

<b>Pros and Cons</b>

<table>
<tr><td>Pros</td><td>Cons</td></tr>
<tr><td>Simpler structure with fewer classes.</td><td>Violates the Open/Closed Principle because new spawn types would require changes to the core Spawner class.</td></tr>
<tr><td>Easier to maintain in smaller projects</td><td>Violates the Single Responsibility Principle by handling multiple entity types in one class.</td></tr>
</table>

<b>Finalised Solution</b>

I implemented a <b>Spawner</b> interface to handle the logic for enemy spawning. Each entity, like <b>ManFly</b> and <b>Spirit</b>, has its own dedicated spawner class (<b>ManFlySpawner</b> and <b>SpiritSpawner</b>). The <b>Graveyard</b> class, which extends <b>Ground</b>, works with these spawners to control when and where enemies appear. The <b>ManFlySpawner</b> and <b>SpiritSpawner</b> classes implement the <b>Spawner</b> interface, ensuring that enemies are placed in the correct locations in the game. This setup keeps the spawning system flexible, allowing for easy addition of new enemies without disrupting existing mechanics.

<b>Reason for Decision:</b>
1. Single Responsibility Principle (SRP): Each spawner class handles spawning for a specific entity (like ManFlySpawner or SpiritSpawner).
2. Open/Closed Principle (OCP): New spawner types (like VampireSpawner) can be added without modifying existing spawner classes.
3. Interface Segregation Principle (ISP): The Spawner interface is simple, with one clear purpose—spawning entities.
4. Dependency Inversion Principle (DIP): The Graveyard depends on the Spawner interface, not specific spawner</td>
</tr>
<tr>
<td>Created

Graveyard class

(extends Ground)</td>
<td><b>Role &amp; Responsibility:</b>

Represents a unique type of ground where enemy spawning, occur.

interacts with spawner classes to manage the appearance of enemies like ManFly and Spirit.

<b>Relationship:</b>
- Extends Ground.
- Used by Spawner implementations to determine the spawning location of new enemies</td>
</tr>
<tr>
<td>Created

ManFlySpawner class

(implements Spawner)</td>
<td><b>Role &amp; Responsibility:</b>

Handles the spawning of <b>ManFly</b> enemies, ensuring they appear in appropriate game locations.

<b>Relationship:</b>
- Implements Spawner.
- Spawns ManFly enemies into the game.
- Used by Graveyard or other mechanics to manage enemy spawning logic.</td>
</tr>
</table>

Created SpiritSpawner class (implements Spawner)	**Role & Responsibility:** Handles the spawning of **Spirit** enemies, ensuring they appear in appropriate game locations. **Relationship:** - Implements Spawner. - Spawns Spirit enemies into the game. - Used by Graveyard or other mechanics to manage enemy spawning logic.	implementations, making the system more flexible. 5. Liskov Substitution Principle (LSP): Any spawner (e.g., SpiritSpawner, ManFlySpawner) can be swapped into the Graveyard without breaking the game. 6. DRY Principle: The Spawner interface centralizes spawning logic, avoiding repeated code across different entities. **Limitations & Tradeoffs:** 1. More Classes: Each entity requires its own spawner class, adding more classes to the project. 2. Interface Overhead: While using interfaces adds abstraction and might seem complex for smaller projects, it provides scalability and maintainability for larger systems.