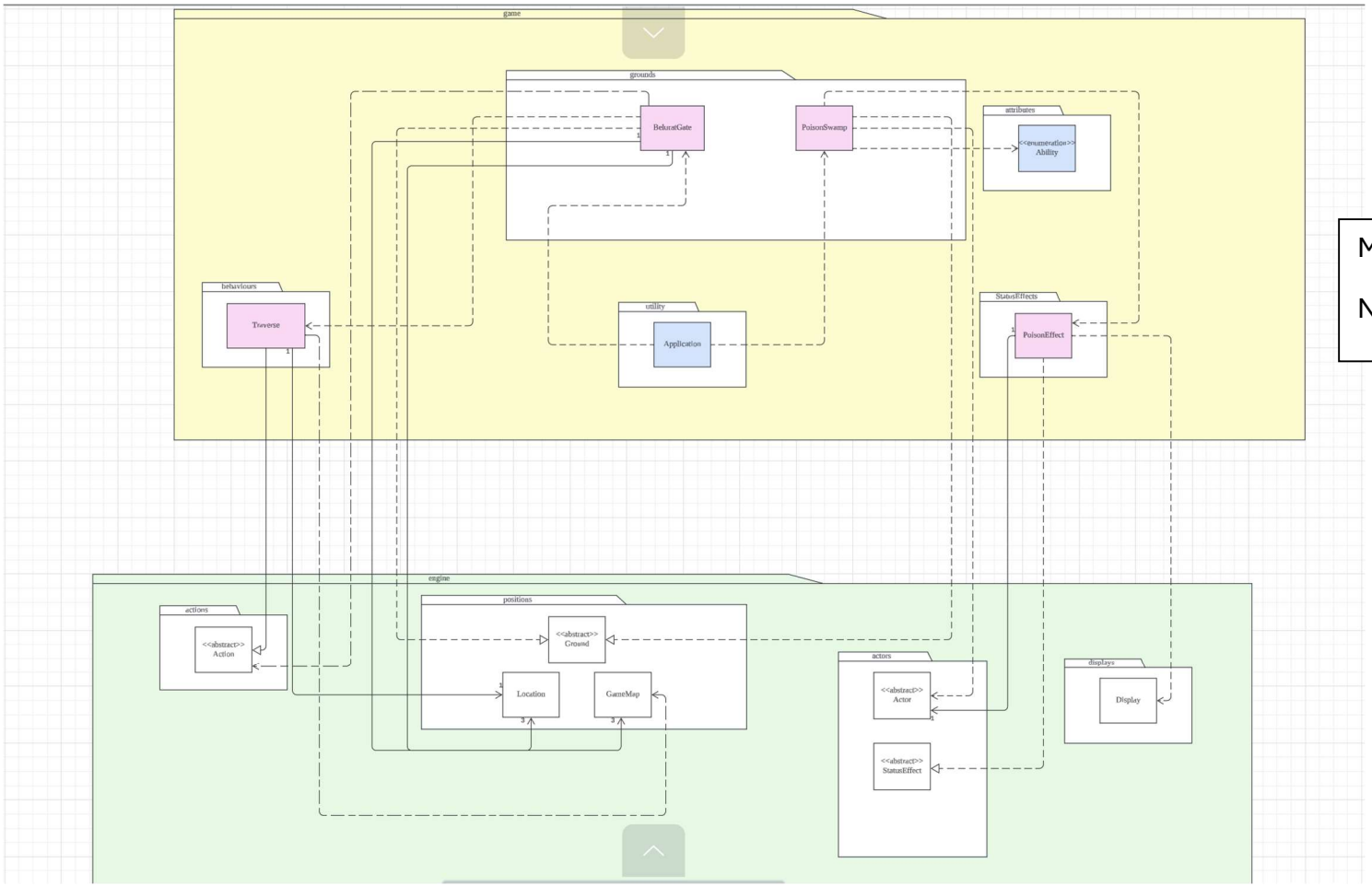
Design Rationale

Req 2 – Belurat, Tower Settlement

UML diagram:



Modified class = Light Blue

New class = Pink

Classes Modified / Created	Roles and Responsibilities	Rationale
BeluratGate (extends Ground)	Class showing BeluratGate Relationships: 1. Extends the “Ground” class for ground functionality	**Alternate Solution:** Create an abstract class “Gate” and make three different concrete class that extends the “Gate” abstract class. Then, implement the details for each gate in each gate concrete subclass respectively **Finalized Solution:** Instead of creating excessive classes, create one concrete class “BeluratGate” and implement all details for the gates in this class. **Reasons for Decision:** 1. **Single Responsibility Principle:** BeluratGate only handles one responsibility that is to represent the gate properties and it does not handle other responsibility (Traverse logic). 2. **Open-closed Principle:** The BeluratGate class is designed to be open for extension but closed for modification. By using the TraverseAction, we can easily extend the functionality of the gate without modifying its existing

		code. This allows us to add new types of actions or destinations in the future without changing the core logic of the gate. 3. **Liskov substitution Principle:** "BeluratGate" extends the "Ground" and provides specific implementations for different ground types. They inherit the base functionality from "Ground" and can be used interchangeably with "Ground" without affecting the program. 4. **Interface Segregation Principle** By adhering to ISP, "BeluratGate" only depends on the "Ground" abstract, which is relevant to their functionality. Moreover, "BeluratGate" are not forced to depend on interfaces they do not use. This improves the readability of the code and makes the code more focused. 5. **Dependency Inversion Principle**

		The “BeluratGate” class depends on the abstraction of the “Actions” class rather than a concrete implementation. This allows for greater flexibility and easier testing, as different actions can be injected into the gate without changing its implementation. 6. **Don’t Repeat Yourself** By combining all the details into one concrete “BeluratGate” class, avoids code repetition, as the initialization of the gates are actually the same. Limitations and Tradeoffs: 1. The use of multiple abstract classes can increase the complexity of the codebase, making it harder to understand and maintain 2. Debugging issues in a highly abstracted and modular codebase can be more complex, it requires a deep

		understanding of the interactions between different components. 3. The use of multiple layers of abstraction (abstract classes) can introduce overhead when understanding and navigating the codebase.
Traverse (extends Action)	Class to show traverse action from one location to another location Relationships: 1. Extends Action abstract class to have an "execute()" method and "menuDescription()" method to ensure that the traverse action is executed and shows in the menu	**Alternate solution:** Create an abstract "TraverseAction" that extends "Action". Then implement each traverse action for each particular gate. **Finalized solution:** Instead of creating excessive classes, create one concrete class "Traverse" and implement traverse logic in this class **Reasons for decision:** 1. **Single Responsibility Principle:** The TraverseAction class is designed to have a single responsibility: to handle the traversal of an actor from one location to another. This is to prevent the code from being more complex when other
PoisonSwamp (extends Ground)	Class showing PoisonSwamp Relationships: 1. Extends the "Ground" class for ground functionality	
PoisonEffect (extends StatusEffect)	Class to show poison effect that is applied through PosionSwamp Relationships:	

	1. Extends “StatusEffect” to allow “PoisonEffect” to have status effect functionality	consumables are introduced into the program. **2. Open-closed Principle:** The “PoisonEffect” class can be extended by creating subclasses that add new behavior or modify existing behavior without changing the original “PoisonEffect” class. For example, you could create a “StrongPoisonEffect” class that extends “PoisonEffect” and overrides the tick method to apply more damage. The existing functionality of the “PoisonEffect” class does not need to be modified to add new features. Any new behavior can be added through inheritance. **3. Liskov Substitution Principle (LSP):** The “TraverseAction” class can be used interchangeably with other actions that extend the Action class without affecting the correctness of the program. This ensures that the “TraverseAction” can be used in any context where an Action is expected, maintaining the integrity of the system. **4. Interface Segregation Principle**
Ability	Enum class to represent abilities Relationships: “PoisonSwamp” class uses “POISON_RESISTANT” enum to check if the actor can be poisoned	

		The “TraverseAction” class implements only the methods it needs from the “Action” interface. This ensures that the class is not burdened with unnecessary methods, keeping it focused and efficient 5. **Dependency Inversion Principle:** “Traverse” class depends on the abstract “Action” class and uses abstract interfaces for “Location” and “GameMap”, adhering to the Dependency Inversion Principle by ensuring high-level modules do not depend on low-level modules but on abstractions. 6. **Don’t Repeat Yourself:** By combining all the details into one concrete “Traverse” class, avoids code repetition, as the traverse logic is actually the same. Limitations and Tradeoffs: 1. It could be a significant workload and effort when modifications are needed due to the introduction of new conditions

		2. Using enums could lead to tight coupling between classes, as they all depend on the same enum definitions. For example, if “Ability” enum is modified, all classes that uses it has to be modified again, which increases burden of maintaining.