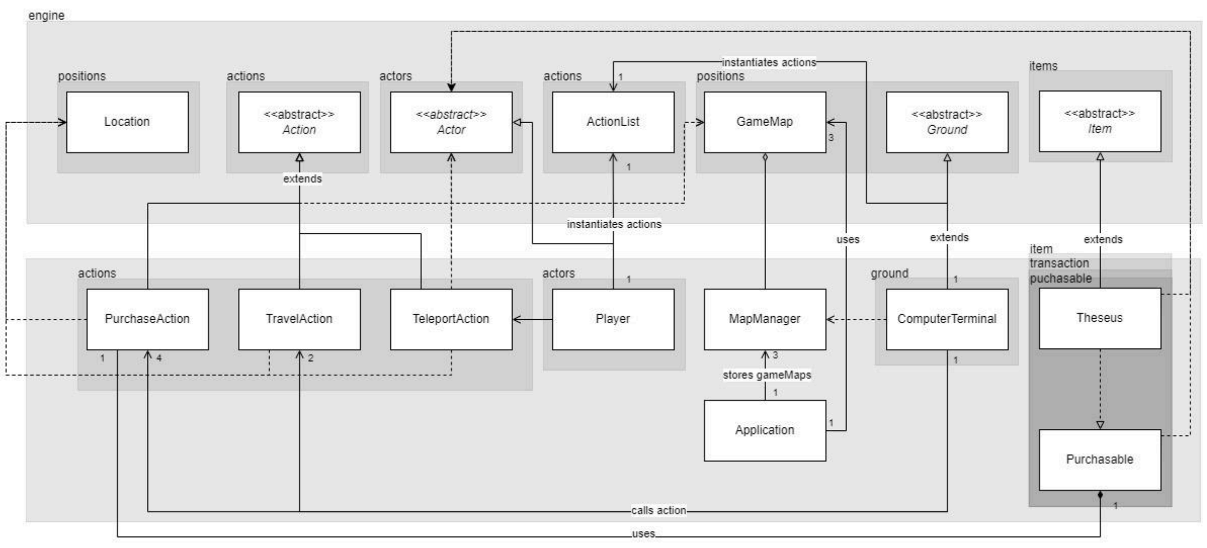
Assignment 3 Design Rationale

REQ1 : The Ship of Theseus

UML



Design Goal

The goal is to enhance the gameplay experience by providing:

- A mechanism for the player to travel between different maps using computer terminals.
- A special item (Theseus) that allows the player to teleport within the same map.

Changes Made

Introduced Purchasable Interface:

To promote better encapsulation and adhere to the Single Responsibility Principle, Introduced a Purchasable interface. This interface encapsulates all purchase-related methods, separating the responsibility for handling purchases from other classes."

Item-Specific Purchase Logic:

Moved the item-specific purchase logic from the PurchaseAction class to the respective item classes like EnergyDrink, DragonSlayerSwordReplica, and Theseus. This change further aligns with the Single Responsibility Principle.

Updated PurchaseAction:

The PurchaseAction class was updated to use the Purchasable interface for executing purchases, promoting polymorphism and making the code more flexible and maintainable.

Modified ComputerTerminal:

Modified the ComputerTerminal class to use Purchasable items for purchase actions.

Design Decision

ComputerTerminal Class

- Responsibility: Allows the player to interact with it to travel to different maps.
- Design Choice: ComputerTerminal maintains a list of all GameMap instances and their names, and generates TravelAction instances to provide travel options.
- Justification: Centralizes the travel functionality, making it easy to manage and extend.

TravelAction Class

- Responsibility: Facilitates the travel between maps.
- Design Choice: TravelAction takes a GameMap instance and places the player at a specified location on the target map.
- Justification: Provides a clear and straightforward way to implement map transitions, ensuring the player moves between maps seamlessly.

Theseus Item

- Responsibility: Allows the player to teleport within the same map.
- Design Choice: Theseus is an item that, when used, triggers a TeleportAction.
- Justification: Adds an element of strategic movement and surprise, enhancing gameplay dynamics.

TeleportAction Class

- Responsibility: Executes the teleportation of the player to a random location within the current map.
- Design Choice: TeleportAction calculates a random location within the current map's boundaries and moves the player to that location.
- Justification: Simplifies the teleportation logic and keeps the player's movements within the same map, maintaining game balance.

MapManager Class

- Responsibility: Manages game maps and their names.
- Design Choice: Utilizes static methods and lists for centralized management.
- Justification: Simplifies map handling, ensures global accessibility, and enhances maintainability.

Alternative Design

Embed Map Information within Each Computer Terminal. Each ComputerTerminal instance could maintain its own list of maps it can travel to, excluding the current one.

Pros: Simple to implement, easy to manage maps at the Computer Terminal level.

Cons: Redundant information, harder to update if new maps are added.

Final Design

Encapsulation

Each class encapsulates its own functionality. For example, **TravelAction** encapsulates the logic for moving between maps, and **TeleportAction** encapsulates the logic for teleporting within a map. The **MapManager** class encapsulates the logic for managing maps and their names, providing a clean interface for other parts of the application.

Single Responsibility Principle

Each class has a single responsibility: **ComputerTerminal** for providing travel options, **TravelAction** for handling map transitions, **Theseus** for enabling teleportation, and **TeleportAction** for executing the teleportation.

Open/Closed Principle

The design is open for extension. For instance, new types of actions or items can be added without modifying existing code. The **ComputerTerminal** can be extended to offer new services without altering its core functionality.

Polymorphism

Actions like **TravelAction** and **TeleportAction** inherit from a common Action base class, allowing them to be used interchangeably where actions are expected. This leverages polymorphism to treat different types of actions uniformly.

Pros & Cons

Pros

- **Separation of Concerns**: Clear separation between map management and travel logic.
- **Encapsulation**: The map name retrieval logic is encapsulated within MapManager.
- **Flexibility**: Easy to extend with additional maps or actions.
- **Maintainability**: Adheres to OOP principles, making the code easier to maintain and understand.

Cons

- **Tight Coupling**: The TravelAction is tightly coupled with the GameMap, which may make it harder to change the travel logic without affecting other parts of the system.
- **Complexity**: Slight increase in complexity due to additional class and methods.

Conclusion

The final design effectively integrates the new travel and teleportation features into the game, enhancing the player's experience while maintaining a clear and manageable code structure. By adhering to object-oriented principles such as encapsulation, the single responsibility principle, and polymorphism, the design ensures that the system is both flexible and robust, capable of accommodating future expansions and modifications with minimal disruption.