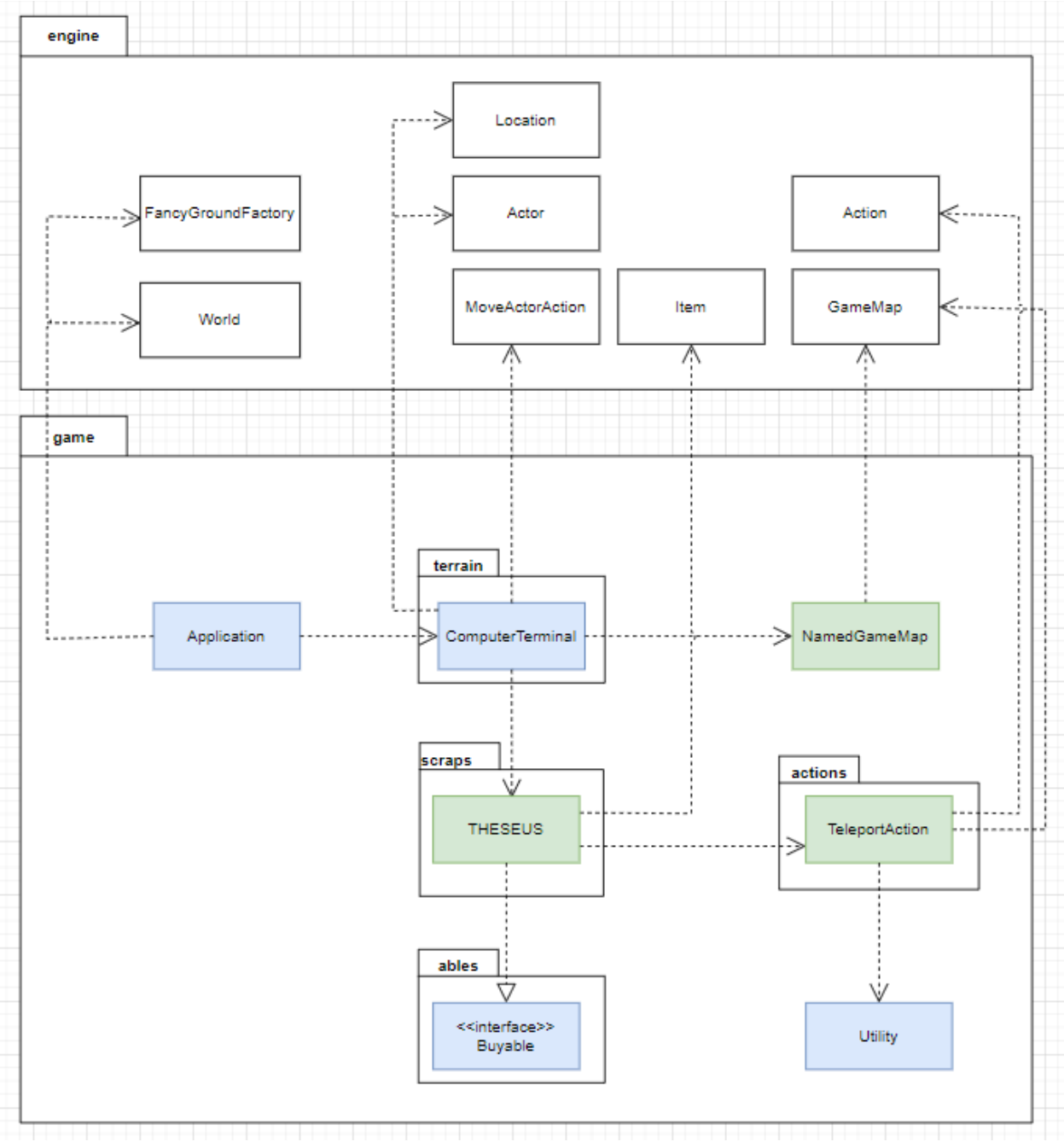
Assignment 3 Design Rationale

Requirement 1: The Ship of Theseus

UML Diagram:



Design Goal:
The design goal for this assignment is to implement a teleportation feature and an item purchasing system within the game, optimizing the player's interaction with various elements in the game world. This involves designing classes that handle item transactions and player teleportation, adhering closely to SOLID principles and best practices to ensure the system is maintainable, scalable, and efficient.

Design Decision:
In implementing the teleportation feature and item purchasing system, the decision was made to introduce the THESEUS, TeleportAction, and ComputerTerminal classes. The THESEUS class represents a teleportation item that players can purchase and use. The TeleportAction class handles the action of teleporting an actor to a random location within the same map. The ComputerTerminal class allows players to buy items and teleport between different maps.

Rationale:
Efficiency: By encapsulating teleportation and item purchasing logic within distinct classes, we ensure that each class has a single responsibility, making the code easier to maintain and extend.

Scalability: The design allows for easy addition of new items and actions without modifying existing code, adhering to the Open/Closed principle.

Maintainability: The clear separation of concerns makes the system easier to debug and update, reducing the likelihood of introducing bugs when changes are made.

Alternative Design:
One alternative approach could involve merging the teleportation and purchasing functionalities into a single class or directly embedding these features into the player class. For example, the player class could have methods for both buying items and teleporting.

Snippet of Alternative Design:

```
public class Player extends Actor {
   // Other attributes and methods...

   public String buyItem(Item item, int price) {
      if (this.getBalance() < price) {
         return this + " does not have enough credits to buy " + item;
      }
      this.deductBalance(price);
      this.addItemToInventory(item);
      return this + " bought " + item + " for " + price + " credits";
   }

   public String teleport(GameMap map) {
      int x = Utility.generateNumber(0, map.getXRange().max());
      int y = Utility.generateNumber(0, map.getYRange().max());
      Location newLocation = map.at(x, y);
      if (newLocation.canActorEnter(this)) {
         map.moveActor(this, newLocation);
         return this + " is teleported to a new location.";
      } else {
         return this + " tried to teleport, but the teleportation failed.";
      }
   }
}
```

Analysis of Alternative Design:
The alternative design is not ideal because it violates various Design and SOLID principles:

Single Responsibility Principle (SRP):

- **Violation:** The Player class handles multiple responsibilities: player actions, buying items, and teleporting.
- **Impact:** This makes the Player class more complex and harder to maintain.

Open/Closed Principle (OCP):

- **Violation:** Adding new items or actions requires modifying the Player class.
- **Impact:** This increases the risk of introducing bugs and makes the system less flexible to changes.

Interface Segregation Principle (ISP):

- **Violation:** The Player class may end up with too many methods, some of which might not be relevant for all players.
- **Impact:** This makes the class interface less clear and harder to use correctly.

Final Design:
In our design, we closely adhere to relevant design principles and best practices, such as SOLID and DRY:

Single Responsibility Principle (SRP):

- **Application:** Each class has a single responsibility: THESEUS for item representation and purchasing, TeleportAction for teleporting, and ComputerTerminal for handling item transactions and map teleportation.

- **Why:** Adhering to SRP ensures that each class is focused and easier to understand, maintain, and test.

- **Pros/Cons:**
  - **Pros:** Simplified class structure, easier maintenance.
  - **Cons:** More classes to manage, but this is outweighed by the benefits.

Open/Closed Principle (OCP):

- **Application:** The design allows adding new items and actions without modifying existing classes. New items can implement the Buyable interface and new actions can extend the Action class.

- **Why:** This promotes extensibility and reduces the risk of introducing errors when new features are added.

- **Pros/Cons:**
  - **Pros:** Flexible and scalable design.
  - **Cons:** Requires careful design of interfaces and abstract classes.

Interface Segregation Principle (ISP):

- **Application:** Interfaces and abstract classes are designed to be small and focused. For example, Buyable only includes methods related to buying items.

- **Why:** This ensures that classes implementing these interfaces are not burdened with irrelevant methods.

- **Pros/Cons:**
  - **Pros:** Clear and focused interfaces, easier to implement and understand.
  - **Cons:** Requires more interfaces, but this complexity is manageable.

Conclusion:

Overall, our chosen design provides a robust framework for implementing the teleportation and item purchasing features. By carefully considering relevant design principles, we have developed a solution that is efficient, scalable, and maintainable. This design paves the way for future enhancements, such as adding new types of items and actions, without requiring significant changes to the existing codebase.