DRY (Don't Repeat Yourself)

- TeleportAction.java: The execute method checks if the destination contains an actor and if not, moves the actor to the destination. This logic is not repeated anywhere else in the code, adhering to the DRY principle.

- ComputerTerminal.java: The addDestinations method adds a destination to the destinations map. This method is used instead of directly accessing and modifying the destinations map in multiple places

SOLID Principles

- The Open-Closed Principle (OCP) is also demonstrated in the design. For instance, if a new character or item is added in the future, existing classes do not need to be modified. New classes can be created that implement the necessary interfaces, ensuring that existing code is closed for modification but open for extension.

- Liskov Substitution Principle (LSP): The code adheres to LSP as the subclasses can be substituted for their base classes. For example, TeleportAction can be used wherever Action is expected.

- The `TeleportCapable` interface is a clear example of the Interface Segregation Principle (ISP). They provide very specific capabilities that classes can implement, ensuring that a class only needs to implement the methods that are relevant to it.

- Dependency Inversion Principle (DIP): High-level modules (Theseus, ComputerTerminal) do not depend on low-level modules (TeleportAction). Both depend on abstraction (TeleportCapable). This principle leads to a system that is more decoupled and thus easier to refactor, change, and deploy.

Connascence

Connascence of Algorithm: The addDestinations method in the TeleportCapable interface doesn't impose any specific algorithm for adding destinations. The actual algorithm is defined in the implementing classes (Theseus and ComputerTerminal), reducing the likelihood of errors due to changes in the algorithm.

Advantages of the Design

- **Polymorphism**
  The TeleportCapable interface allows for polymorphism. Any class that implements this interface can be treated as a TeleportCapable object, making the code more flexible and extensible. If a new class needs to have teleportation capability, it can simply implement this interface.

- **Dependency Injection**
  In Application.java, the dependency injection of game maps and terminals is done by creating instances of GameMap and ComputerTerminal, and then adding them to the world and locations respectively. This allows for easy addition or removal of maps and terminals, and the specific details of each map or terminal are encapsulated within their respective classes, promoting high cohesion and low coupling.

Disadvantages of the Design:

Theseus.java and ComputerTerminal.java: Both classes implement the TeleportCapable interface and have their own addDestinations method. This is a repetition of code and violates the DRY principle.

Also, the destination field in TeleportAction is directly accessed and modified, which breaks encapsulation.

Ways to Extend the Design:

If we need to add a new type of ground that allows a player to teleport, like ComputerTerminal, we can create a new class for this ground type and have it implement the TeleportCapable interface.