REQ1: Design Rationale (Teleportation)

A. Structuring Teleportation Mechanisms

Design A1: Hardcoded teleportation logic per mechanism type

Implement separate, standalone teleportation methods within each teleportation class (TeleDoor, TeleportationCircle, TeleportCube). Each class contains its own complete teleportation flow, destination management, and side-effect handling (burning, malfunctioning, etc.).

Pros	Cons
Straightforward to implement each mechanism in isolation.	Severe DRY violation: teleportation core logic (moving actors, validating destinations) duplicated across three classes.
Easy to trace the complete behavior of each teleportation type within a single file.	High coupling between teleportation flow and side-effects makes it difficult to modify base teleportation without affecting all mechanisms.
	Violates SRP: each class handles both teleportation coordination and mechanism-specific side effects.
	Closed to extension: adding new teleportation types requires reimplementing the entire teleportation pipeline.

Design A2: Abstract teleportation with specialized side-effect handlers

Introduce a base TeleportAction class that encapsulates core teleportation logic (destination validation, actor movement, occupied location checks). Each mechanism (TeleDoor, TeleportationCircle, TeleportCube) extends TeleportAction through inner action classes that override execute() to inject mechanism-specific behavior

Pros	Cons
High cohesion: core teleportation logic lives in one reusable action class.	Adds a layer of abstraction requiring subclass coordination.

DRY principle maintained: destination validation and actor movement are not repeated.	
Open for extension: new teleportation mechanisms only need to implement their unique side-effects without touching core logic.	
Clear separation of concerns: TeleportAction handles movement, subclasses handle mechanism-specific consequences.	

The project adopts Design A2 because it properly separates the invariant teleportation mechanics (moving actors between locations, checking for occupied destinations, validating same-location teleports) from variant side-effects (burning patterns, malfunctions).

B. Managing Destinations and Multi-Target Selection

Design B1: Fixed single-destination per mechanism

Each teleportation mechanism stores exactly one destination. If multiple destinations are needed, create multiple instances of the same mechanism type at the same location.

Pros	Cons
Minimal data structure: just store one Location reference.	Poor user experience: cluttered map with multiple overlapping mechanisms.
Straightforward implementation with no selection logic needed.	Violates realism and design intent: a single door/circle should offer multiple destinations.
	Increases map complexity unnecessarily and makes testing cumbersome.

Design B2: Multi-destination support with dynamic action generation

Each teleportation mechanism maintains a list of TeleportDestination objects. When the player interacts with the mechanism, it generates multiple TeleportAction instances via allowableActions(), one per destination. The player selects their desired destination from the menu.

Pros	Cons
Natural user interface: player sees all available destinations and chooses.	Slightly more complex destination management (requires a collection).

Single mechanism instance can serve multiple purposes, reducing map clutter.	
Consistent with engine design patterns: leveraging allowableActions for player choice.	
Scalable: adding destinations doesn't require code changes, just configuration.	

The implementation follows **Design B2** because it provides a clean, extensible way to manage multiple destinations while maintaining a good user experience. The engine's allowableActions pattern naturally supports this design, and it avoids polluting the map with redundant mechanism instances.

C. Implementing Mechanism-Specific Side Effects

Design C1: Side-effects embedded in mechanism Ground classes

Each Ground subclass (TeleDoor, TeleportationCircle) directly implements its burning logic within its own methods. The burning patterns are hardcoded into the mechanism's teleportation flow.

Pros	Cons
Side-effects are "close to" their trigger point.	Tight coupling between terrain representation and action execution.
No additional action classes needed.	Violates SRP: Ground classes should represent terrain, not execute complex action side-effects.
	Makes testing difficult: cannot test burning logic independently from terrain.
	Breaks engine conventions: Ground classes should primarily handle terrain state, not perform actions.

Design C2: Mechanism-specific TeleportAction subclasses with overridden execute()

TeleDoor and TeleportationCircle create inner action classes that extend TeleportAction. These action subclasses override execute() to call super.execute() for core teleportation, then apply their specific burning patterns:

- TeleDoor: burns all adjacent tiles at the destination

- TeleportationCircle: burns one random adjacent tile at the source before teleporting

Pros	Cons
Clear separation: Ground classes generate actions, actions perform effects.	Requires creating specialized action classes for each mechanism.
Testable: burning logic can be tested independently through action execution.	
Maintains engine conventions: Ground classes remain focused on terrain state and action generation.	
Flexible: easy to modify burning patterns without changing core teleportation or terrain classes.	
Reusable burning logic: FireGround and Dirt terrain classes can be used by any system	

The design uses Design C2 because it properly respects the engine's architectural boundaries. Ground classes handle terrain state and expose affordances through allowableActions, while Action classes handle behavior execution. This separation makes the system more maintainable and keeps each class focused on its primary responsibility. The implementation correctly places burning side-effects in the execute() method of the action subclasses, not in the Ground classes themselves.

D. Handling Teleport Cube Malfunction and Inventory Constraints

Design D1: Malfunction logic in TeleportCube item class

The TeleportCube item class directly implements the 50% malfunction chance and random location selection within the item's methods, checking inventory status and executing teleportation inline.

Pros	Cons
All cube-specific logic in one place.	Violates SRP: Item class handles both item state and complex action execution.
	Tight coupling: item must know about maps, random location selection, and movement mechanics.

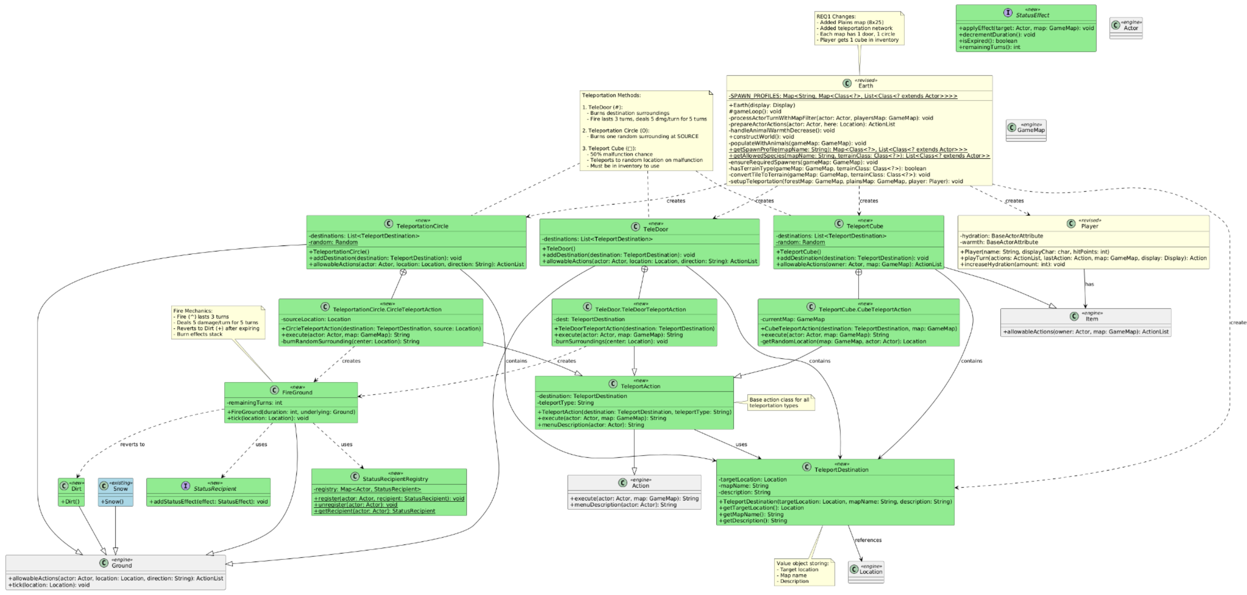
	Difficult to reuse malfunction logic or random teleportation for other items.
	Breaks engine patterns: items should expose affordances, not execute actions.

Design D2: CubeTeleportAction with inventory-gated affordance

TeleportCube overrides allowableActions() to return actions only when the item is in the actor's inventory. The cube generates specialized CubeTeleportAction instances that extend TeleportAction. CubeTeleportAction's execute() performs the 50% malfunction check first: if it malfunctions, it finds a random valid location on the current map and moves the actor there directly; otherwise it calls super.execute() for normal teleportation to the selected destination.

Pros	Cons
Clean separation: item manages state and affordance conditions, action manages execution.	Requires implementing random location selection utility within the action.
Consistent with engine design: items generate actions, actions perform behavior.	
Testable: malfunction probability and random selection can be unit tested.	
Inventory constraint naturally enforced through allowableActions pattern.	
Random location logic encapsulated in action, reusable for other teleportation items.	

The implementation adopts **Design D2** because it maintains proper separation of concerns between items and actions. The item class focuses on state management and condition checking (is it in inventory?), while the action class handles the complex teleportation logic with malfunctions. This keeps the codebase aligned with engine conventions and makes the malfunction behavior easy to test and modify.


REQ1 Changes:
- Added Plains map (8x25)
- Added teleportation network
- Each map has 1 door, 1 circle
- Player gets 1 cube in inventory
«new» StatusEffect
+applyEffect(target: Actor, map: GameMap): void
+decrementDuration(): void
+isExpired(): boolean
+remainingTurns(): int
«engine» Actor
Teleportation Methods:
1. TeleDoor (#):
- Burns destination surroundings
- Fire lasts 3 turns, deals 5 dmg/turn for 5 turns
2. Teleportation Circle (O):
- Burns one random surrounding at SOURCE
3. Teleport Cube (□):
- 50% malfunction chance
- Teleports to random location on malfunction
- Must be in inventory to use
«revised» Earth
-SPAWN_PROFILES: Map<String, Map<Class<?>, List<Class<? extends Actor>>>>
+Earth(display: Display)
#gameLoop(): void
-processActorTurnWithMapFilter(actor: Actor, playersMap: GameMap): void
-prepareActorActions(actor: Actor, here: Location): ActionList
-handleAnimalWarmthDecrease(): void
+constructWorld(): void
-populateWithAnimals(gameMap: GameMap): void
+getSpawnProfile(mapName: String): Map<Class<?>, List<Class<? extends Actor>>>
+getAllowedSpecies(mapName: String, terrainClass: Class<?>): List<Class<? extends Actor>>
-ensureRequiredSpawners(gameMap: GameMap): void
-hasTerrainType(gameMap: GameMap, terrainClass: Class<?>): boolean
-convertTileToTerrain(gameMap: GameMap, terrainClass: Class<?>): void
-setupTeleportation(forestMap: GameMap, plainsMap: GameMap, player: Player): void
«engine» GameMap
creates
creates
creates
creates
creates
«new» TeleportationCircle
-destinations: List<TeleportDestination>
-random: Random
+TeleportationCircle()
+addDestination(destination: TeleportDestination): void
+allowableActions(actor: Actor, location: Location, direction: String): ActionList
«new» TeleDoor
-destinations: List<TeleportDestination>
+TeleDoor()
+addDestination(destination: TeleportDestination): void
+allowableActions(actor: Actor, location: Location, direction: String): ActionList
«new» TeleportCube
-destinations: List<TeleportDestination>
-random: Random
+TeleportCube()
+addDestination(destination: TeleportDestination): void
+allowableActions(owner: Actor, map: GameMap): ActionList
«revised» Player
-hydration: BaseActorAttribute
-warmth: BaseActorAttribute
+Player(name: String, displayChar: char, hitPoints: int)
+playTurn(actions: ActionList, lastAction: Action, map: GameMap, display: Display): Action
+increaseHydration(amount: int): void
has
«engine» Item
+allowableActions(owner: Actor, map: GameMap): ActionList
Fire Mechanics:
- Fire (^) lasts 3 turns
- Deals 5 damage/turn for 5 turns
- Reverts to Dirt (+) after expiring
- Burn effects stack
«new» TeleportationCircle.CircleTeleportAction
-sourceLocation: Location
+CircleTeleportAction(destination: TeleportDestination, source: Location)
+execute(actor: Actor, map: GameMap): String
-burnRandomSurrounding(center: Location): String
«new» TeleDoor.TeleDoorTeleportAction
-dest: TeleportDestination
+TeleDoorTeleportAction(destination: TeleportDestination)
+execute(actor: Actor, map: GameMap): String
-burnSurroundings(center: Location): void
«new» TeleportCube.CubeTeleportAction
-currentMap: GameMap
+CubeTeleportAction(destination: TeleportDestination, map: GameMap)
+execute(actor: Actor, map: GameMap): String
-getRandomLocation(map: GameMap, actor: Actor): Location
creates
contains
creates
contains
contains
creates
«new» FireGround
-remainingTurns: int
+FireGround(duration: int, underlying: Ground)
+tick(location: Location): void
«new» TeleportAction
-destination: TeleportDestination
-teleportType: String
+TeleportAction(destination: TeleportDestination, teleportType: String)
+execute(actor: Actor, map: GameMap): String
+menuDescription(actor: Actor): String
Base action class for all teleportation types
reverts to
uses
uses
uses
«new» Dirt
+Dirt()
«existing» Snow
+Snow()
«new» StatusRecipient
+addStatusEffect(effect: StatusEffect): void
«new» StatusRecipientRegistry
-registry: Map<Actor, StatusRecipient>
+register(actor: Actor, recipient: StatusRecipient): void
+unregister(actor: Actor): void
+getRecipient(actor: Actor): StatusRecipient
«engine» Action
+execute(actor: Actor, map: GameMap): String
+menuDescription(actor: Actor): String
«new» TeleportDestination
-targetLocation: Location
-mapName: String
-description: String
+TeleportDestination(targetLocation: Location, mapName: String, description: String)
+getTargetLocation(): Location
+getMapName(): String
+getDescription(): String
references
Value object storing:
- Target location
- Map name
- Description
«engine» Location
«engine» Ground
+allowableActions(actor: Actor, location: Location, direction: String): ActionList
+tick(location: Location): void

The diagram above shows the final design of requirement 1, which utilizes each Design 2 approach from parts A, B, C, and D to implement multiple teleportation mechanisms (TeleDoor, TeleportationCircle, TeleportCube) with distinct side-effects and a unified teleportation action framework.

The TeleportAction base class extends the engine's abstract Action class and encapsulates core teleportation logic (actor movement, destination validation, same-location checking, occupied location detection). Mechanism-specific subclasses override execute() to inject their unique side-effects. TeleportDestination acts as a value object carrying location and descriptive metadata. Each mechanism (TeleDoor and TeleportationCircle as Ground subclasses, TeleportCube as an Item subclass) maintains a list of destinations and generates appropriate action instances through allowableActions.

Since teleportation mechanisms share common movement logic but differ in their consequences (burning patterns, malfunction behavior), it makes sense to abstract the core teleportation flow into a base class. This avoids repeated code (DRY principle) and keeps each mechanism focused on its specific effects (Single Responsibility Principle).

Using the engine's allowableActions pattern to expose teleportation options and separating side-effect execution into action subclasses maintains clear architectural boundaries. Ground classes handle terrain state and action generation; Action classes handle behavior execution; Item classes manage inventory state and conditional affordance exposure. This separation results in better maintainability and safer extensions (Single Responsibility Principle). By keeping the malfunction logic, burning patterns, and destination management separate from core teleportation, we avoid bloated classes and keep each component focused on its purpose (Interface Segregation Principle).

TeleportAction depends on the TeleportDestination abstraction rather than concrete Location internals. The burning mechanics depend on FireGround and Dirt terrain classes rather than hardcoding fire behavior into teleportation actions. This removes direct dependencies from actions to specific map implementations and prevents scattering teleportation logic across the codebase. When introducing new teleportation mechanisms (for example, a Warp Stone or Portal Scroll) or modifying burning patterns, we don't need to change the base TeleportAction or existing mechanisms—new classes simply extend the action and implement their own side-effects (Open-Closed Principle). The high-level teleportation flow depends on abstractions (TeleportDestination, Location, GameMap) while low-level details (specific coordinates, map layouts, random location selection) live behind these abstractions (Dependency Inversion Principle).