Syed Meekal Hussain Zaidi – 32846606

A1 Design Rationale

Requirement 1

BroadSword (Child of TradeableWeaponItem):

- Inheritance: BroadSword effectively utilizes inheritance, inheriting behavior from TradeableWeaponItem, which adheres to the DRY (Don't Repeat Yourself) principle.
- Open-Closed Principle (OCP): The class extends the parent class without modifying it, aligning with the OCP.
- Pros:
  - Reuses code from the parent class for common weapon attributes.
  - Enhances code maintainability and readability through inheritance.
- Cons:
  - None apparent in this specific class.
- Extendability:
  - Easy to add new actions or attributes specific to BroadSword without altering the parent class.

FocusAction (Child of TickableAction):

- Inheritance: FocusAction inherits from TickableAction, following the DRY principle by reusing common action execution and tracking logic.
- Pros:
  - Encapsulates the logic for activating and deactivating the Focus skill, promoting the Single Responsibility Principle (SRP).
  - Provides clear menu descriptions and outcomes for the action.
- Cons:
  - The use of a global Display instance for printing messages could potentially lead to issues with extensibility and unit testing.
- Extendability:
  - Easy to customize FocusAction behavior without affecting the parent class.

TickableAction (Abstract Base Class):

- Inheritance and Polymorphism: Serves as an abstract base class for tickable actions, reducing code duplication and promoting the DRY principle.
- Pros:
  - Defines a common interface for tickable actions, facilitating the implementation of various actions with similar behavior.
  - Enforces the use of a tick mechanism, ensuring actions are executed over multiple game rounds.
- Cons:
  - The concrete methods **execute** and **menuDescription** are not used in this abstract class and could be considered unnecessary.
- Extendability:
  - Allows the creation of various tickable actions by extending this base class.

General Code Design Considerations:

- SOLID Principles:
  - The code demonstrates adherence to SOLID principles by encapsulating behavior within classes and promoting inheritance and polymorphism.
- DRY Principle:
  - Code duplication is minimized through inheritance and the use of a base class for tickable actions.
- Pros:
  - The code structure is well-organized and modular, making it easier to understand and extend.
  - Adherence to SOLID principles promotes maintainability and scalability.
- Cons:
  - The code uses a global **Display** instance for printing messages, which could lead to issues with testing and future extensibility.

Extendability:

- The code is designed to be easily extensible, allowing for the addition of new weapon types and tickable actions without significant modifications to existing classes.
- Potential areas for improvement include decoupling the display mechanism and using dependency injection for better testability and flexibility.

Requirement 2

Implementation & Why:

- Spawner (Child of Ground):
  - Adherence to SOLID and DRY:
    - Encapsulates the logic for spawning enemy actors in a separate class, promoting the Single Responsibility Principle (SRP).
    - Inherits from Ground, which is a common base class for terrain, reducing redundancy and adhering to the DRY principle.
  - Pros:
    - Encapsulates the spawning logic, making it modular and easy to reuse for different types of spawners.
    - Allows for the dynamic spawning of enemy actors based on probabilities and conditions.
  - Cons:
    - None apparent in this specific class.
  - Extendability:
    - Easy to create various types of spawners by extending this base class.

- Void (Child of Ground):
    - Adherence to SOLID and DRY:
        - Inherits from Ground for a common base class for terrain, adhering to the DRY principle.
    - Pros:
        - Implements a specific behavior for the Void ground, making it clear and self-contained.
    - Cons:
        - None apparent in this specific class.
    - Extendability:
        - Can be extended to handle additional behaviors or interactions with actors stepping on the Void.
- EnemyActor (Child of Actor):
    - Adherence to SOLID and DRY:
        - Implements specific enemy actor behaviors (e.g., unconscious, dropping items, runes) while extending the Actor class, adhering to the Open-Closed Principle (OCP).
        - Encapsulates various enemy actor-related attributes and behaviors.
    - Pros:
        - Encapsulates enemy actor behaviors, promoting the Single Responsibility Principle (SRP).
        - Provides a flexible framework for creating diverse enemy actors with different behaviors and item drops.
    - Cons:
        - The class has multiple attributes and behaviors, which might become complex for highly specialized enemy actors.
    - Extendability:
        - Allows for the creation of a wide range of enemy actors by extending this base class.
        - Customizable behaviors and item drops make it adaptable to various game scenarios.
- Graveyard (Child of Spawner):
    - Adherence to SOLID and DRY:
        - Extends the Spawner class, which encapsulates spawning logic, adhering to the Open-Closed Principle (OCP).

        - Inherits attributes and behaviors from the Spawner class, promoting code reuse and adhering to the DRY principle.
    - Pros:
        - Specializes the Spawner class for a specific type of spawner (graveyard), making it clear and focused.
        - Utilizes the existing spawner framework for dynamic enemy actor spawning.
    - Cons:
        - None apparent in this specific class.
    - Extendability:
        - Easy to create other types of spawners by extending the Spawner class.
        - Customizable spawning behavior and actor types for different spawner instances.

Pro:

- Modularity: The code is organized into separate classes for different purposes, making it modular and easier to maintain.
- Code Reuse: Inheritance and encapsulation are used effectively, reducing code duplication and promoting the DRY principle.
- Flexibility: EnemyActor and Spawner classes provide a framework for creating diverse enemy actors and spawners with different behaviors and characteristics.
- Transparency: The code is clear and self-contained, making it easy to understand and modify.

Cons:

- Complexity: EnemyActor class can become complex for highly specialized enemy actors, potentially leading to increased maintenance effort.

Extendability:

- Spawner Class: Provides a foundation for creating various types of spawners, enabling dynamic enemy actor spawning in different game scenarios.
- EnemyActor Class: Enables the creation of a wide range of enemy actors with customized behaviors, item drops, and runes.
- Graveyard Class: Specializes the spawner behavior for graveyard-specific enemy actor spawning, allowing for different types of spawners to be created by extending the Spawner class.

Requirement 3

Implementation & Why:

Created AttackBehaviour:

- An AttackBehaviour class has been created to encapsulate the behavior of attacking a nearby player.
- AttackBehaviour class implements the Behaviour interface and provides the getAction method to create an AttackAction when a player is detected nearby.
- This design follows the Single Responsibility Principle (SRP) as the behavior for attacking is separated from the actor class.

Modified WanderingUndead:

- The WanderingUndead class now includes the AttackBehaviour.

Modified Floor

- Override Ground's canActorEnter() method to only allow Actors with CAN_ENTER_FLOOR capabilities
- Use of enum means there is no need for 'instance of', following OCP.

Modified Player

- Added CAN_ENTER_FLOOR capability

Pros:

- Creation fo AttackBehaviour separates the attacking behavior, making it reusable for other actors with similar behavior.
- The Floor class efficiently checks an actor's capability using the Status enum, reducing code duplication and promoting maintainability.
- WanderingUndead can attack any actor with HOSTILE_TO_ENEMY capability.

Cons:

- AttackBehaviour code is somewhat similar to WanderBehaviour code, both using for loops for Exit and location information. A parent class that defines for loop implementation was possible, but would require many if/else statements to cater to AttackBehaviour and WanderBehaviour's different returning Action types.

Extendability

- More actors can be given the CAN_ENTER_FLOOR capability, allowing them to enter Floor ground types without changing code in Floor

Requirement 4

Implementation & Why:

- **Gate (Child of Ground):**
  - **Adherence to SOLID and DRY:**
    - Inherits from Ground for a common base class for terrain, adhering to the DRY principle.
    - Implements specific logic for locked gates, promoting the Single Responsibility Principle (SRP).
  - **Pros:**
    - Implements a clear and specific behavior for locked gates, enhancing code readability.
    - Supports gate unlocking, allowing actors to pass through.
  - **Cons:**
    - None apparent in this specific class.
  - **Extendability:**
    - Can be extended to handle additional gate-related features or behaviors.
- **TravelAction (Child of Action):**
  - **Adherence to SOLID and DRY:**
    - Encapsulates the logic for traveling to another map in a separate class, promoting the Single Responsibility Principle (SRP).
  - **Pros:**
    - Encapsulates travel logic, making it modular and easy to reuse for various game scenarios.
    - Provides a clear interface for actors to transition between maps.
  - **Cons:**
    - None apparent in this specific class.
  - **Extendability:**
    - Can be used to facilitate travel between different game maps, making it suitable for multi-map scenarios.
- **OldKey (Child of Item):**
  - **Adherence to SOLID and DRY:**
    - Inherits from Item for a common base class for game items, adhering to the DRY principle.

    - **Pros:**
        - Represents a simple game item (old key) with minimal complexity.
    - **Cons:**
        - Limited functionality; it only serves as an example item.
    - **Extendability:**
        - Can be extended to create various types of game items with additional features.
- **UnlockAction (Child of Action):**
    - **Adherence to SOLID and DRY:**
        - Encapsulates the logic for unlocking gates in a separate class, promoting the Single Responsibility Principle (SRP).
    - **Pros:**
        - Encapsulates gate unlocking logic, making it modular and easy to reuse.
        - Clearly defines the action for unlocking a gate and checks for the presence of the old key.
    - **Cons:**
        - Specific to gate unlocking and may not be suitable for other types of actions.
    - **Extendability:**
        - Can be used as a template for creating other item-based actions, such as opening chests or doors.

Pro:

- **Modularity:** The code is organized into separate classes, each responsible for a specific aspect, enhancing modularity.
- **Code Reuse:** Encapsulated logic can be easily reused for various game scenarios.
- **Clarity:** The code clearly defines gate-related actions and interactions, improving code readability.

Cons:

- **Specificity:** Some classes are highly specialized for particular behaviors (e.g., UnlockAction for gate unlocking), limiting their versatility.

Extendability:

- **Gate Class:** Can be extended to handle additional gate-related features or behaviors, such as different types of gates or interactions.
- **TravelAction Class:** Provides a foundation for handling travel between maps, facilitating multi-map scenarios.

- **OldKey Class:** Can be extended to create various types of game items with different attributes and functionalities.
- **UnlockAction Class:** Can serve as a template for creating other item-based actions, expanding its use beyond gate unlocking.

Requirement 5

Implementation and Why:

- **Gate (Child of Ground):**
  - **Adherence to SOLID and DRY:**
    - Inherits from Ground for a common base class for terrain, adhering to the DRY principle.
    - Implements specific logic for locked gates, promoting the Single Responsibility Principle (SRP).
  - **Pros:**
    - Implements a clear and specific behavior for locked gates, enhancing code readability.
    - Supports gate unlocking, allowing actors to pass through.
  - **Cons:**
    - None apparent in this specific class.
  - **Extendability:**
    - Can be extended to handle additional gate-related features or behaviors.
- **TravelAction (Child of Action):**
  - **Adherence to SOLID and DRY:**
    - Encapsulates the logic for traveling to another map in a separate class, promoting the Single Responsibility Principle (SRP).
  - **Pros:**
    - Encapsulates travel logic, making it modular and easy to reuse for various game scenarios.
    - Provides a clear interface for actors to transition between maps.
  - **Cons:**
    - None apparent in this specific class.
  - **Extendability:**
    - Can be used to facilitate travel between different game maps, making it suitable for multi-map scenarios.
- **OldKey (Child of Item):**
  - **Adherence to SOLID and DRY:**
    - Inherits from Item for a common base class for game items, adhering to the DRY principle.
  - **Pros:**
    - Represents a simple game item (old key) with minimal complexity.
  - **Cons:**
    - Limited functionality; it only serves as an example item.
  - **Extendability:**

        - Can be extended to create various types of game items with additional features.
- **UnlockAction (Child of Action):**
    - **Adherence to SOLID and DRY:**
        - Encapsulates the logic for unlocking gates in a separate class, promoting the Single Responsibility Principle (SRP).
    - **Pros:**
        - Encapsulates gate unlocking logic, making it modular and easy to reuse.
        - Clearly defines the action for unlocking a gate and checks for the presence of the old key.
    - **Cons:**
        - Specific to gate unlocking and may not be suitable for other types of actions.
    - **Extendability:**
        - Can be used as a template for creating other item-based actions, such as opening chests or doors.

Pro:

- **Modularity:** The code is organized into separate classes, each responsible for a specific aspect, enhancing modularity.
- **Code Reuse:** Encapsulated logic can be easily reused for various game scenarios.
- **Clarity:** The code clearly defines gate-related actions and interactions, improving code readability.

Cons:

- **Specificity:** Some classes are highly specialized for particular behaviors (e.g., UnlockAction for gate unlocking), limiting their versatility.

Extendability:

- **Gate Class:** Can be extended to handle additional gate-related features or behaviors, such as different types of gates or interactions.
- **TravelAction Class:** Provides a foundation for handling travel between maps, facilitating multi-map scenarios.
- **OldKey Class:** Can be extended to create various types of game items with different attributes and functionalities.
- **UnlockAction Class:** Can serve as a template for creating other item-based actions, expanding its use beyond gate unlocking.