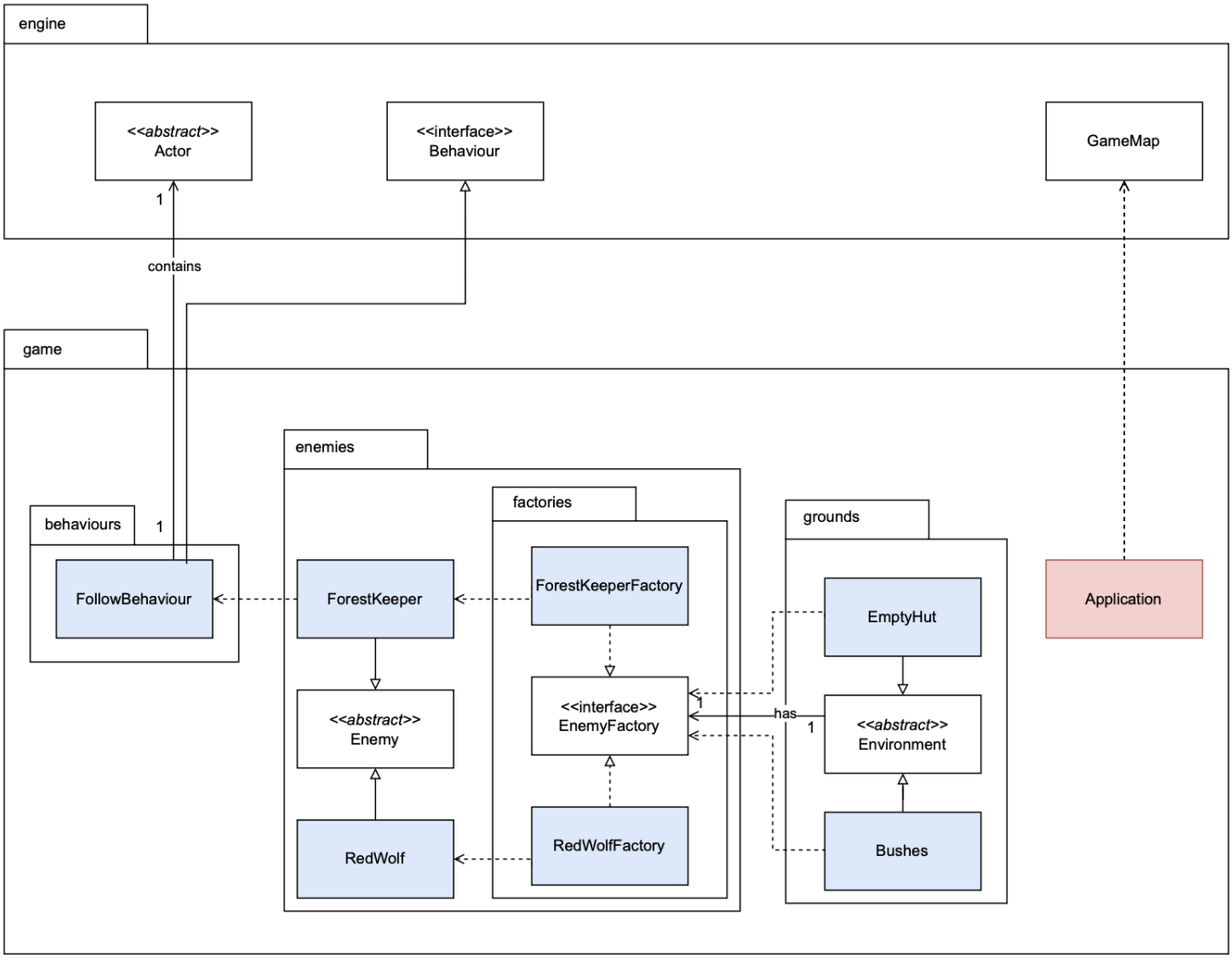
Design Rationale (REQ 1)



Changes Made from Assignment 1

In the original design of the Application class from Assignment 1, the initialization of maps, population of actors, and allocation of items were all confined within the main function. Such a structure posed potential challenges for scalability, especially when the game were to be expanded with additional maps and allocations in Assignment 2.

In response, the code is restructured to introduce specific, dedicated methods for each initialization and population task. For instance, game maps now have their initialization handled by methods like initializaAbandonedVillage and initializeBurialGround. Similarly, entities and features specific to each map are managed by methods such as populateAbandonedVillage and populateBurialGround.

This refactoring aligns with the Single Responsibility Principle (SRP) by ensuring that each method handles a distinct aspect of the game setup. Moreover, it upholds Don't Repeat Yourself (DRY) principle, preventing the repetition of similar patterns for different game features and promoting code reusability. These changes not only enhance the clarity and readability of the code but also ensure its maintainability and extensibility.

Implementation

Classes Added

1. ForestKeeper extends Enemy
2. RedWolf extends Enemy
3. EmptyHut extends Environment (which implements Spawnable interface)
4. Bushes extends Environment (which implements Spawnable interface)
5. ForestKeeperFactory extends EnemyFactory
6. RedWolfFactory extends EnemyFactory
7. FollowBehaviour extends Behaviour

Codes Modified

1. A new game map, Ancient Woods, was added in the main Application.
2. Two locked gates were added (one in Ancient Woods and one in Burial Ground), connecting both maps.
3. For each enemy added:
    - Its name, display character, hit points, and spawn rate were set.
    - Its intrinsic weapon was set.
4. For each environment added:
    - Their respective enemyFactory was set, which spawns enemies of a particular type within the ground.
    - The location of the ground is ticked every game turn to spawn enemies based on their spawn rates.
5. The FollowBehaviour was implemented in such a way that enemies can follow the player if the player is within their vicinity. Here is how the follow behaviour works:
    - An Actor (the follower) follows another Actor (the target).
    - The getAction method determines the first optimal direction (exit) for the follower to move to get closer to the target.
    - The optimal direction is determined based on the Manhattan distance between the follower and the target.
    - If such a direction is identified, the method returns a MoveActorAction representing the move; otherwise, it returns null, indicating no action.
    - The class provides functionality for enemies to chase or track the player based on their positions on a game map.
6. FollowBehaviour is added to the enemies in Ancient Woods (Forest Keeper and Red Wolf) through their allowableActions method. Here:
    - FollowBehaviour is given the highest priority among all other enemy behaviors (AttackBehaviour and WanderBehaviour).
    - This means that if the player keeps moving, the enemies will follow the player instead of attacking him.
    - AttackBehaviour is activated once the player stops moving, and they are within the exits of the enemy.8. Modify the unconscious method inside the Forest Keeper and Red Wolf class such that they drop healing vial upon death (caused by another actor).
7. The unconscious method in the Forest Keeper and Red Wolf classes was modified so that they drop a healing vial upon death (caused by another actor).

Design Principles

Interface EnemyFactory

This interface was created to represent specific ground objects' capability of spawning actor objects.

Design Principles Followed

1. **Interface Segregation Principle (SOLID)**
    - The EnemyFactory interface ensures that the classes that implement it provide a specific functionality, i.e, creating an enemy. This allows for flexible expansion with new enemy types.
2. **Open-Closed Principle (SOLID)**
    - Each factory implementation like ForestKeeperFactory and HollowSoldierFactory can create a specific type of enemy without modifying the EnemyFactory interface. If a new enemy type is to be added, only a new factory class needs to be created, and the interface remains unmodified.
3. **Single Responsibility Principle (SOLID)**
    - Each factory has one clear responsibility - to produce a specific type of enemy. This principle promotes cohesion within the classes, ensuring that they only handle one specific task.
4. **Loose coupling**
    - The factory pattern used here ensures a degree of decoupling. Instead of the game instantiating enemies directly, it delegates this responsibility to the factory classes. This means changes in enemy instantiation won't directly affect the game's main logic.
5. **DRY (Don't Repeat Yourself)**
    - By using factories, the instantiation logic of each enemy type is kept in one place. For instance, if the way a ForestKeeper is instantiated changes, it only needs to be updated in the ForestKeeperFactory class and not in every place the enemy is created in the game.
6. **High Cohesion**
    - Each factory class is highly cohesive because it's solely focused on creating a specific enemy type. There's no mixing of functionalities or responsibilities, making the code easier to maintain and understand.

Cons of this Design

1. **Abstraction Overhead**
    - Introducing an interface for enemy creation adds another level of abstraction, which might not always be necessary if there are only a few enemy types or if the instantiation process is straightforward.
2. **Maintenance Overhead**
    - Any changes or expansions to the enemy creation process would mean adjustments to the interface, potentially affecting all factories implementing it.
3. **Less Intuitive**
    - For someone new to the codebase, understanding the need for factories might not be immediately clear. They might wonder why direct instantiation isn't used, leading to a slight learning curve.

Abstract Class Enemy

This abstract class was created to represent any non-player actors.

Design Principles Followed

1. **Single Responsibility Principle (SOLID)**
    - The Enemy abstract class primarily handles the functionalities related to an enemy, such as adding the capability to perform attack action to actors which are hostile to enemies, adding capability to confirm that they are enemies, and also determining their spawn rate.
2. **Open-Closed Principle (SOLID)**
    - The Enemy class adds a new layer of abstraction to represent all enemies. This class contains all required attributes that non-player characters have (behaviours, spawn rate, etc.). This means any extensions of the system involving the addition of new types of enemies can be achieved by the addition of new classes that extend the Enemy class without a need to modify the Enemy class.
3. **Liskov Substitution Principle (SOLID)**
    - The subclasses (ForestKeeper and RedWolf) are substitutable for their base class (Enemy). This is evident in methods like allowableActions where the subclasses call the superclass's method via super.allowableActions() and then add their unique behaviours.
4. **Dependency Inversion Principle (SOLID)**
    - The Enemy class does not reply on concrete implementations of behaviours. Instead, it has a Map of behaviours that it iterates over, allowing for flexibility in adding or changing behaviours without altering the core Enemy logic.
5. **DRY (Don't Repeat Yourself)**
    - The common functionalities and properties among enemies are kept in the Enemy class, ensuring there's no repetition. Specific functionalities, such as dropping items upon death, are defined in subclasses as needed.
6. **High Cohesion**
    - The Enemy class focuses solely on functionalities related to enemies. This design ensures that related functionalities are bundled together in a logical manner.
7. **Low Coupling**
    - The behaviours like AttackBehaviour and WanderBehaviour are most likely encapsulated in their own classes, which means that the Enemy class isn't tightly coupled with the specific implementations of these behaviours. Instead, it just relies on the contract (abstract class) that these behaviours provide.

Cons of the Design

1. **Tight Coupling with Behaviours**
    - The Enemy class directly instantiates certain behaviors within its constructor, namely AttackBehaviour and WanderBehaviour. This reduces the flexibility for subclasses to define their own sets of behaviors without overriding or modifying the parent class. However, for the current state, every enemy attacks and wander around, and hence we will keep it as so for now.
2. **Direct Dependency on Concrete Items**
    - The ForestKeeper subclass directly depends on concrete items like HealingVial and Runes. This is a sign that the abstract class might not be abstracting enough details, leading to potential tight coupling between enemies and specific game items.

Follow Behaviour

The FollowBehaviour class implements a behavior where an actor tries to move closer to a target actor.

Design Principles Followed

1. **Single Responsibility Principle (SOLID)**
    - The FollowBehaviour class has a single responsibility: making an actor follow another actor. All the methods and properties within the class support this sole functionality.
2. **Liskov Substitution Principle (SOLID)**
    - This class implements the Behaviour interface, which ensures that wherever a Behaviour is needed, a FollowBehaviour can be used interchangeably without causing any issues.
3. **High Cohesion**
    - This class is dedicated to providing a following mechanism. It doesn't try to implement unrelated behaviours or functionalities, ensuring it has high cohesion.
4. **Low Coupling**
    - It mainly depends on the Behaviour interface, Actor, and Location, keeping its dependencies minimal and focused on its purpose.
5. **DRY (Don't Repeat Yourself)**
    - The method centralizes the logic for calculating distance, ensuring that whatever distance needs to be calculated within the class, this method can be called, preventing repetition.

Cons of the Design

1. **Lack of Consideration for Multiple Paths**
    - The algorithm only considers the first valid location it finds that reduces the distance to the target. It doesn't necessarily find the best path or the shortest path to the target.

ForestKeeperFactory and RedWolfFactory

The classes are responsible for creating and returning a new instance of their particular enemies (Forest Keeper and Red Wolf).

Design Principles Followed

1. **Single Responsibility Principle (SOLID)**
    - Each factory class (ForestKeeperFactory and RedWolfFactory) has a single responsibility: creating an instance of its respective enemy.
2. **Open-Closed Principle (SOLID)**
    - Both classes are open for extension (we would extend them to add more initialization parameters for the enemies in the future), but closed for modification (we won't have to change the classes when adding new types of enemies)
3. **Liskov Substitution Principle (SOLID)**
    - Both classes implement the EnemyFactory interface, meaning wherever EnemyFactory is expected, any of these factory classes can be used interchangeably.
4. **Interface Segregation Principle (SOLID)**
    - They adhere to a simple interface (EnemyFactory) that expects only a method to create an enemy. This means that the classes aren't burdened with methods they don't use.
5. **Don't Repeat Yourself (DRY)**
    - The enemy creation logic is encapsulated within the respective factory. If there were any complex initialization or setup steps for creating enemies, they would be centralized here, ensuring no repetition across the codebase.
6. **Low Coupling**

- The factories are loosely coupled. If you were to change the internals of RedWolf or ForestKeeper, you wouldn't necessarily have to change the factories, as they depend on the abstract Enemy type.

Cons of this Design

1. **Overengineering for Simplicity**
   - If the creation of enemies like ForestKeeper and RedWolf remains as straightforward as just calling their constructors, these factory classes might seem like overkill. Direct instantiation might suffice.
2. **More Classes to Manage**
   - With each new enemy type, a new factory class needs to be introduced. As the number of enemies grows, managing these factories could become cumbersome.

EmptyHut and Bushes

The classes EmptyHut and Bushes represent specific types of environments within a game. They have the capability to spawn enemies at regular intervals.

Design Principles Followed

1. **DRY (Don't Repeat Yourself)**
   - The tick method has been centralized in the parent Environment class. This ensures that the logic for spawning an enemy on each game tick is defined in one place, avoiding code duplication. Both EmptyHut and Bushes inherit this behaviour, ensuring that modifications in enemy spawning behaviour will only need to be made in one place in the future.
2. **Single Responsibility Principle (SOLID)**
   - EmptyHut and Bushes have a clear and single responsibility: to represent specific environments in the game world. All the logic concerning enemy spawning is encapsulated within the Environment class, ensuring that EmptyHut and Bushes only need to focus on their unique characteristics.
3. **Open/Closed Principle (SOLID)**
   - Environment class is open for extension (child classes can still add more specific behavior if needed) but closed for modification (enemy spawning behavior can be consistent across all child classes).
4. **Liskov Substitution Principle (SOLID)**
   - The EmptyHut and Bushes can be used interchangeably with their parent Environment class without affecting the behavior. This is because they inherit and do not change the core functionality provided by the parent.
5. **High Cohesion**
   - High cohesion is achieved by ensuring that each class is focused on a specific task. The Environment class handles the generic behaviour related to any game environment that can spawn enemies, while EnptyHut and Bushes define specific types of environments without delving into the details of enemy spawning.
6. **Low Coupling**
   - Low coupling is maintained by using the EnemyFactory interface. Both EmptyHut and Bushes are not tightly coupled to a specific enemy creation logic but instead rely on the EnemyFactory interface. This means changes to specific enemy creation details won't affect these classes.

Cons of the Design

1. **Tight Coupling with Enemies**
    - The Environment class (from which Bushes and EmptyHut inherit) is tightly coupled with the enemy creation mechanism due to the EnemyFactory dependency. This could be problematic if in the future there's a need for these grounds to spawn other non-enemy entities or if the enemy creation process changes dramatically. However, based on our current requirement, all the spawnable grounds spawn enemies and not spawning any other non-ememy actors, we would keep the implementation like this.
2. **Memory Overhead**
    - The instantiation of the Random class within the Environment class might introduce memory overhead, especially if there are numerous instances of Bushes and EmptyHut on the map.

ForestKeeper and RedWolf

Design Principles Followed

1. **Single Responsibility Principle (SOLID)**
    - Both the ForestKeeper and RedWolf classes primarily represent specific enemy types in the game. They encapsulate behavior related to those specific enemies, such as their intrinsic weapon, the items they drop upon death, and the actions they can perform.
2. **Open-Closed Principle (SOLID)**
    - Both classes are open for extension (can be subclassed to create variations) but closed for modification. Their core behaviors can be extended by overriding methods, but the core structure remains unchanged.
3. **Liskov Substitution Principle (SOLID)**
    - Since ForestKeeper and RedWolf extend the Enemy class, they are expected to be substitutable for Enemy. Their overridden methods are consistent with the behavior expected from the Enemy class.
4. **High Cohesion**
    - High cohesion is maintained in both classes. All the methods and properties within each class revolve around a single concept: defining the behaviour and attributes of the respective enemy type.

Pros of the Design

1. **Abstraction**
    - Introduction of the new layer of abstraction; abstract class Enemy and interface EnemyFactory not only makes the system easy to comprehend, but also to extend. Any addition of new enemy types can be followed by addition of its own factory object.

Cons of the Design

1. **Duplicate Code**
    - Both classes have similar implementations for the methods unconscious and allowableActions. However, other enemies might drop different items upon death, and also have different behaviours to be added using allowableActions method, their implementations would still be different from each other.
2. **Future Complexity**

- The number of enemy factory classes grows linearly with the number of enemy types. This might introduce complexities within the system as the number of child classes of Enemy grows large in the future.