FIT2099 Assignment 3: DESIGN RATIONALE

By: Fahad Assadi, Ishan Ingolikar, Debashish Sahoo, Kevin Chen

REQ 1

Design Choice #1

The features to visit the new map and to spawn the enemies (where we have spawners for each type of enemy) are pretty much identical to the approach used in Assignment 1. Huts and Bushes extend an abstract EnemyNest class, which in turn extends the Ground class. Each EnemyNest stores a Spawner interface type object. Spawner classes exist for all types of enemy subclasses, whose job is to simply return a new instance of the relevant enemy. In the tick method of the EnemyNest abstract class, the spawnActor() method of the spawner is called if the probability hits. Creating the enemies is also the same as previous assignments where we make the enemy subclass extend the Enemy abstract class.

Now, while the feature for visiting a new map is the same as before, which is to pass a MoveActorAction to the LockedGate, now in order to accommodate for a single LockedGate to allow travel to multiple locations, we pass in an array of MoveActorActions to a LockedGate while instantiating it. Inside the LockedGate class, we just loop through and add all the MoveActorActions.

Pros

- The modification of passing in an array of MoveActorActions to the LockedGate constructor as opposed to passing one earlier is a **simple** and intuitive approach that adds **minimal complexity** to the code and easily achieves the desired result.
- The feature to store an array of MoveActorActions in LockedGate adheres to the **Open-Closed Principle (OCP)**, as if we wanted a certain locked gate to move to a third location in the future, we'd can easily pass in more MoveActorActions in the array in the Application class.
- Though we considered making the Spawner class an abstract class extending Ground instead of being an interface and having Spawner implement the spawning logic in its tick() method, having the tick() method in the EnemyNest abstract class allows us to reduce repetition by implementing and triggering the spawn in one place, as it's common for all "enemy nests", thus adhering to the **Do Not Repeat Yourself Principle (DRY).**

Cons

- Deciding against making the Spawner class an abstract class extending Ground instead of being an interface and having Spawner implement the spawning logic in its tick() method, the additional layer of **complexity** rising from the EnemyNest abstract class which connects the Ground to the Spawner persists.

Decision

Apart from considering changing the Spawner implementation, there weren't any alternative approaches that sprung up during the design process, and weighing the pros and cons, we were happy to stick with this design choice.

REQ 2

Design Choice #1

The Blacksmith was implemented such that the once the player entered the vicinity of the Blacksmith, the blacksmith object would get the inventory of the player and iterate over each item to check if it can be upgraded by. Two interfaces, Upgrade and LimitedUpgrade were used to force all items to have either the upgrade() or the limitedUpgrade() functionality based on the upgradelimit which would be an intrinsic attribute. When the players inventory was scanned through/iterated through, the blacksmith would try and **downcast** the item into an Upgradable or a LimitedUpgradable and would call the .upgrade() or .limitedUpgrade()

Pros

- **Interface Segregation Principle** is adhered to as if a new LimitedUpgradable item is added it can implement only the LimitedUpgradable interface.

- **Single Repsonsibility maintained** by each item that has its own logic for upgrade.

- **Open and Closed Principle** adhered to as a new Upgradable item can be easily added by just implementing the interface.

Cons

- **Liskov Substitution Principle** is not adhered to and leads to **massive downcasting** which is a major **code smell.**

- An **unnecessary Interface** is added, to **Upgrade** an item is just one peripheral functionality (regardless of how many times you can do the same.) and so it does not seem prudential or intuitive to have more than one interface dedicated to the same functionality.

Design Choice #2

In this design, we take a different approach. Instead of the checking when the player is in the Blacksmith's vicinity we take advantage of the allowableActions(Actor otherActor, Location location) method of the **Item**, this method allows the item holder to perform an action on the neighbouring 'otherActor' by the virtue of the item. Thus, in this method we just check based on the (Ability.UPGRADES which the Blacksmith has) to check if the neighbouring actor is the Blacksmith, if it is then, we check if this.upgradable() is true (boolean returning function) if it is, we display a UpgradeAction ready to be selected as a menu option to the player.

We have one interface, **Upgradable** that all upgradable items like Broadsword and Healing Vial implement. How we implement the concept of One time Upgrades (LimitedUpgrades) earlier is by having a private static final variable called DEFAULT_UPGRADE_LIMIT which is set to 1 for LimitedUpgradable items and Integer.MAX_VALUE for items that can be upgradable without restriction.

This interface forces the concrete contractual implementations of three functions:

1) upgrade() – Carries out the upgrade logic for that item with **Single Responsibility** and adds one to the upgrade counter.

2) isUpgradable() – Returns if the upgrade counter is still less than the DEFAULT_UPGRADE_LIMIT.

3) getUpgradePrice() – Returns the price for the items upgrade.

The UpgradeAction just executes upgrade() if isUpgradable() returns true and deducts the getUpgradePrice() amount from the player's balance (only if the player has enough runes obviously)

Pros

- **Single Repsonsibility maintained** by each item that has its own logic for upgrade**.** **UpgradeAction** handles the execution of the upgrade itself.

- **Liskov Substitution Principle** is adhered to as when the UpgradeAction is executed, the item in question **polymorphs** to the higher abstraction of "Upgradable" allowing full control over the methods related to upgrade.

- **Interface Segregation Principle** is adhered to as if a new Upgradable item is added it can implement only the Upgradable interface and the upgrade restrictions can be added locally (**SRP** adhered to again).

- **Open and Closed Principle** adhered to as a new Upgradable item can be easily added by just implementing the interface.

Cons

- **Harder to understand**, because of the merging of the LimitedUpgrade and Upgrade interface functionality from earlier into just one interface, the code becomes harder to follow and comprehend. Items polymorphing to Upgradables is also hard to keep track of.

- **Narrative Inconsistency**, the Blacksmith is no longer executionally involved in the upgrade itself. Because, the upgrade happens within Upgradable items' allowableActions() and has nothing to do with the Blacksmith class. (More than needed autonomy, **Single Responsibility** is not adhered to here, because not only does the item define its own logic of upgrade() but also queue itself to be upgraded as well)

Decision

Taking into accounts both these design choices, we decided to implement the second design as the code smells of **downcasting** as well as the existence of two interfaces for one functionality seemed to outweigh the benefits of the first design.

REQ 3

We took into consideration 2 different approaches to implement the “Conversation” feature, where different actors can converse with the player. The basic idea consistent across all approaches is having a Listenable interface that is implemented by all actors that can converse and a Listen Action that takes a Listenable actor and returns the message from the monologue list of the Listenable actor.

Design Choice #1

We start with a Listenable Interface that is implemented by all actors that can be conversed with. This Listenable interface has a method called getMonologueList that returns a list of different monologue options that the actor can have. These monologue options can be added based on certain conditions. Then we have a Listen Action that has an association with a listenable actor and calls the getMonologueList method. Here we have Blacksmith that implements the Listenable interface and defines the getMonologueList method with monologue options specific to the blacksmith.
We then have an events singleton class that contains booleans for all events that occur in the game (such as boss deaths, game phases to hardmode, etc). Since the blacksmith monologues that he only converses under special circumstances, the blacksmith accesses the events singleton and checks the booleans in the class. It then returns the monologue list with all the different options to the Listen Action. We then return a random message from the monologue list.

Pros

- The events are all centralised incase we have more features in the future that have conditions based on certain events. **(Open Closed Principle)**
- The listenable interface is responsible for managing monologue options, while the Events singleton class handles event management. **(Single Responsibility Principle)**
- The use of a listenable interface segregates the functionality, making the codebase cleaner and aligning with the Interface Segregation Principle.
- The Listen Action has an association with Listenable Interface instead of an actor (like blacksmith) directly. **(Dependency Inversion Principle)**

Cons

- The Events singleton class vastly increased the difficulty for new developers. Since managing a growing number of events and conditions can become challenging and requires proper documentation.
- The modularity introduced may lead to unintended bugs, such as incorrect event handling or monologue options not appearing when they should.
- Frequent event and condition checks may introduce performance overhead if not optimised.

Design Choice #2

The second approach we tried to reduce the complexity of the code dramatically. We learnt that status applied to certain items will be inherited by the actor holding the item as well.
So, we did away with the events singleton and added a status to the player for the events (like DEFEATED_ABXERVYER and CARRIES_GREAT_KNIFE). Then in Blacksmith, we check for if the player has these status. The rest of the implementation is the exact same as the first design choice.

Pros

- This design significantly reduces code complexity by eliminating the need for an Events singleton and related event checks.
- By directly checking the player's status, the design can potentially improve performance by avoiding unnecessary event condition checks.
- The player status can be used for new features that might use them for special configurations of classes. **(Open Close Principle)**
- With fewer components involved, there is a reduced potential for bugs related to event management and monologue availability.

Cons

- While this design simplifies event checks, it may introduce tighter coupling between the blacksmith and the player's status. If not managed carefully, this could violate the Dependency Inversion Principle.
- As more events and status conditions are added, managing player statuses might become complex and challenging to maintain.

Decision

Weighing the pros and cons of the different approaches, we were happy to go with **Design Choice #2**, which had the most advantages and least drawbacks with the lowest complexity design.

REQ 4

We took into consideration 2 different approaches to implement the “Conversation” feature, where different actors can converse with the player. The basic idea consistent across all approaches is having a Listenable interface that is implemented by all actors that can conversate and a Listen Action that takes a Listenable actor and returns the message from the monologue list of the Listenable actor.

Design Choice #1

We start with a Listenable Interface that is implemented by all actors that can be conversed with. This Listenable interface has a method called getMonologueList that returns a list of different monologue options that the actor can have. These monologue options can be added based on certain conditions. Then we have a Listen Action that has an association with a listenable actor and calls the getMonologueList method. Here we have Isolated Traveller that implements the Listenable interface and defines the getMonologueList method with monologue options specific to the Isolated Traveller.
We then have an events singleton class that contains booleans for all events that occur in the game (such as boss deaths, game phases to hardmode, etc). Since the Isolated Traveller monologues that he only converses under special circumstances, the Isolated Traveller accesses the events singleton and checks the booleans in the class. It then returns the monologue list with all the different options to the Listen Action. We then return a random message from the monologue list.

Pros

- The events are all centralised incase we have more features in the future that have conditions based on certain events. **(Open Closed Principle)**
- The listenable interface is responsible for managing monologue options, while the Events singleton class handles event management. (Single Responsibility Principle)
- The use of a listenable interface segregates the functionality, making the codebase cleaner and aligning with the Interface Segregation Principle.
- The Listen Action has an association with Listenable Interface instead of an actor (like Isolated Traveller) directly. **(Dependency Inversion Principle)**

Cons

- The Events singleton class vastly increased the difficulty for new developers. Since managing a growing number of events and conditions can become challenging and requires proper documentation.
- The modularity introduced may lead to unintended bugs, such as incorrect event handling or monologue options not appearing when they should.
- Frequent event and condition checks may introduce performance overhead if not optimized.

Design Choice #2

The second approach we tried to reduce the complexity of the code dramatically. We learnt that status applied to certain items will be inherited by the actor holding the item as well.
So, we did away with the events singleton and added a status to the player for the events (like DEFEATED_ABXERVYER and CARRIES_GIANT_HAMMER). Then in Isolated Traveller we check for if the player has these status. The rest of the implementation is the exact same as the first design choice.

Pros

- This design significantly reduces code complexity by eliminating the need for an Events singleton and related event checks.
- By directly checking the player's status, the design can potentially improve performance by avoiding unnecessary event condition checks.
- The player status can be used for new features that might use them for special configurations of classes. **(Open Close Principle)**
- With fewer components involved, there is a reduced potential for bugs related to event management and monologue availability.

Cons

- While this design simplifies event checks, it may introduce tighter coupling between the Isolated Traveller and the player's status. If not managed carefully, this could violate the Dependency Inversion Principle.
- As more events and status conditions are added, managing player statuses might become complex and challenging to maintain.

Decision

Weighing the pros and cons of the different approaches, we were happy to go with **Design Choice #2**, which had the most advantages and least drawbacks with the lowest complexity design.

REQ 5

We took into consideration 3 different approaches to implement the “Dream” feature, where various game entities get reset/removed when the player dies. The basic idea consistent across all approaches is having a manager class that globally accesses and modifies the entities.

Design Choice #1

The first approach was to have a **singleton** class called **EntityManager**, which stores the player and arrays of Enemies, Bosses, and Locked Gates objects, and also an array of GameMap objects passed in from the Application class, in order to remove actors from the world. Each of these entities, upon instantiation, would register themselves to the EntityManager in their respective constructors. In the EntityManager class, there would be separate methods to reset each entity, such as resetEnemies(), resetBosses(), etc. A resetEntities() method calling these would in turn be called in the player’s unconscious() method.

Pros

- Singleton class allows us to seamlessly **centralise** the responsibility of resetting/removing entities upon player death.

Cons

- Forcing the singleton EntityManager class to implement the entire functionality for actually resetting/removing all the various entities puts too many various responsibilities onto that one class, violating the **Single Responsibility Principle (SRP)**.
- Having a lot of responsibilities in one class leads to the singleton EntityManager class having **too many dependencies** to each entity class (Enemy, LockedGate, GameMap, etc.).
- In the future, if a new entity is subject to being reset, we’d have to create a new array list, new registration method, and new reset methods, which violates the **Open-Closed Principle (OCP).**

Design Choice #2

The second approach was a massive improvement to the first. We introduced two interfaces called *Resettable* and *Removable*, whose reset() and remove(maps) methods would be implemented by various game entities based on whether their attributes get reset or whether they get removed from the map. The reason why they were created as separate interfaces, is to allow the *remove* method to take in an array of game maps from the EntityManager in order to remove relevant actors from the world, while the *reset* method doesn’t need that. All Resettables and Removables would register themselves to the EntityManager, and there would be methods that iterate through the arrays of Resettables and Removables, and call their respective reset and remove methods.

Pros

- Singleton class allows us to seamlessly **centralise** the responsibility of resetting/removing entities upon player death.
- Having two interfaces called Resettable and Removable allows the classes that implement it to have methods relevant to their needs. Entities that get their attributes reset don't need the game maps, while entities that get removed need it in their remove method. Thus, we adhere to the **Interface Segregation Principle (ISP).**
- Allowing the entity classes themselves to have their reset/remove implementation through the use of interfaces adheres to the **Single Responsibility Principle (SRP)**.
- Using interfaces, if other entities in the future are susceptible to being reset/removed, they can simply implement the relevant interface and its methods. Thus, this method is open to extension and closed for modification, adhering to the **Open-Closed Principle (OCP).**
- Using interfaces **reduces a few dependencies** on concrete entity classes compared to the first approach.

Cons

- Though we got rid of dependencies on concrete entity classes and have dependencies on just Resettables and Removables, there still is a **dependency on GameMap** due to the array of game maps passed into the EntityManager from the Application class.
- Two new interfaces, namely Resettable and Removable, add a bit of **complexity** to the code.

Design Choice #3

The final approach added a lot of improvements to the previous approaches. The Removable interface was removed, and we only stuck with a Resettable interface. The reason why we even had two different interfaces was so that the *remove* method in the Resettable interface could take in an array of game maps from the EntityManager in order to remove relevant actors from the world, while the *reset* method doesn't need that. But, to bypass the need to have a list of Game Maps in the EntityManager, we found a way to **remove the dependency** for this. When the reset() method of the Resettable is called, in the class of the entity that can be removed from the map, the reset() method simply adds a **Status.RESET capability** to the actor. This acts like a **messenger** which signifies that entity is subject to currently being "reset". And in the ***playTurn*** method of that actor, which already has the *map* in its parameters, the actor is simply removed from its map.

Pros

- The dependency on the array of GameMaps in the EntityManager is now removed, **limiting dependencies** to just the Resettables.
- Singleton class allows us to seamlessly **centralise** the responsibility of resetting/removing entities upon player death.

- Allowing the entity classes themselves to have their “reset” implementation through the use of interfaces adheres to the **Single Responsibility Principle (SRP)**.
- If other entities in the future are susceptible to being reset, they can simply implement the relevant interface and its methods, allowing for easy extension and adhering to the **Open-Closed Principle (OCP).**

Cons

- If a lot of different entities in the future are susceptible to being removed from the map, there might be a slight bit of repetition where we’d have to check for the Status.RESET capability and remove the actor in the playTurn method of every single relevant entity class, leading to a mild violation of the **Do Not Repeat Yourself Principle (DRY).**

Decision

Weighing the pros and cons of the different approaches, we were happy to stick with **Design Choice #3**, which had the most advantages and least drawbacks.