FIT2099 Assignment 2: 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 identical to the approach used in Assignment 1.

The new design decision is for the following behaviour of the enemies. To implement this, we implemented a FollowBehaviour class, which implements the Behaviour interface, since this action is undertaken by an NPC, just like AttackBehaviour and WanderBehaviour.

Pros

- This approach adheres to the **Single Responsibility Principle** as the FollowBehaviour class has the sole purpose of containing the logic for following another Actor. This allows for reusability as well.
- This design allows us to add new behaviours to enemies in their constructor without modifying existing ones, thus adhering to the **Open-Closed Principle**.
- This approach adheres to the **Dependency Inversion Principle** as the Enemy class relies on abstractions like Behaviour rather than concrete implementations.
- Using a TreeMap to store the behaviours in the Enemy class helps us sort out the behaviours as opposed to a HashMap.

Cons

- Every enemy subclass that has the following behaviour will need to add FollowBehaviour in its allowableActions(), so a bit of repetition however not any real drawbacks.

Decision

Since most of the features were built on top of our design from Assignment 1, there weren't any alternative approaches that sprung up during the design process. Moreover, weighing the pros and cons, we were happy to stick with this design choice.

REQ 2

Design Choice #1

The feature to add the Bloodberry item to the game is exactly identical to how we implemented the HealingVial and RefreshingFlask in Assignment 1, using the Consumable interface and ConsumeAction. The feature to implement the logic of consuming a Puddle is also identical to the remaining consumables, except that it extends from the Ground class. We just had to use a different allowableActions method that allows the player to consume it only when it's on the location of the puddle, as opposed to anytime from their inventory.

The main design decision is tied to the dropping of Runes by enemies. We created the Runes class, which also implements the Consumable interface, and adds the balance to the player when consumed. The Enemy abstract class contains a hashmap with a key-value pair of the DropAction (of the Runes item with the specific amount it can drop) and the drop chance (of the Runes) when it dies.

Pros

- The beauty of the Consumable interface is that it allows us to use it for multiple types of entities in the game when needed — an Item like Bloodberry and Runes, and even a Ground like Puddle. Allowing for easy extension and no requirement for modification, this adheres very well to the **Open-Closed Principle**.
- Having the Enemy abstract class contain a hashmap with a key-value pair of the DropAction and the drop chance, and calling the addDroppableItems method in the constructor of the abstract class allows the subclasses to populate it with their droppable items and hence this adheres to the **DRY Principle** and also the **Open-Closed Principle**, allowing for easy extensibility.
- The unconscious method in the Enemy abstract class solely does the job of looping through the hashmap of droppable items and executing the DropAction based on the probability. This logic is done once and is common for all enemies, hence reducing repetition, adhering to the **DRY Principle**, encapsulating the shared logic and attributes of all enemies, adhering to the **Single Responsibility Principle,** and allowing the Enemy abstract class to refer to any of the enemy subclasses while accessing their droppable items, following the **Liskov Substitution Principle**.

Cons

- Each of the Enemy subclasses have a dependency on the DropAction class, which could have been avoided by passing the required items and their drop chances to the parent Enemy abstract class which could have had a method to create a DropAction and add it to the hashmap. Apart from that, there aren't any other real drawbacks.

Decision

Building most of the features on top of our design from Assignment 1, and improving some aspects of the design, there weren't any alternative approaches that sprung up during the design process. Weighing the pros and cons, we were happy to stick with this design choice.

REQ 3

Design Choice #1

We create an IsolatedTraveller Class which extends the Actor class, indicating that the merchant is an actor in the game world. IsolatedTraveller has a list of all the items he sells and all its quirks for every item. Then any time a player approaches the merchant, they display all items that they are able to sell to the player by calling a for each loop in the allowable actions of the merchant.

A quirk is any action that the merchant might do when selling the item (eg: increasing the price or scamming the actor). We implement a Strategy similar to Pricing Strategy in the bootcamps, where we have an Interface called Quirk to define special behaviours or quirks that merchants can have during transactions. The interface includes methods for selling and purchasing items with associated quirks, along with a method to determine whether the quirk occurs. The interface includes methods for selling, purchasing, and determining if the quirk occurs. We have multiple quirks like Pricing Quirk, Scam Quirk, Rob Quirk and also a No Quirk when the merchant does not perform any special behaviour.

Eg: Pricing Quirk class that implements the Quirk interface and represents a quirk that affects item pricing during transactions based on a given probability and price percentage.It includes methods to adjust item prices when selling and purchasing items. TransactionItem class acts as a wrapper class that represents an item within a transaction, including its original price.

We then have a Sell and Purchase Action classes to handle selling and purchasing items, respectively. Both actions consider the occurrence of quirks when determining the outcome of a transaction.

Pros

- The Isolated Traveller handles all the items that they are selling and every quirk that they may perform on said item. Every type of quirk only handles their own respective functioning making the classes and interfaces have clear and focused responsibilities, making the code easier to understand and maintain. **(SRP)**
- The Isolated Traveller can easily have a new type of quirk he can perform, because you can create new quirk classes by implementing the Quirk interface and new quirks can be added without modifying the interface itself. **(OCP)**
- The IsolatedTraveller class is a subtype of the Actor class and behaves consistently with its parent class. So the IsolatedTraveller can always be replaced with any other actor when high level classes are performing common actor methods on them. All quirks also polymorph into Quirk in Sell Action and Purchase Action so that their common methods can be executed. **(LSP)**
- Since all high level classes rely on abstractions like Actor and Quirk rather than the concrete implementations like IsolatedTraveller and Pricing Quirk, we can change the concrete implementation without causing any harm to the overall game. **(DIP)**

Cons

- Since All Quirks implement quirk that forces the subclasses to implement Both merchant selling and purchasing. We assumed that all methods will have an implementation for the two methods, but we have Rob quirk that doesn't have a use for the Merchant Selling method, which violates **ISP**. A fix for this would be to have two interfaces selling quirk and purchasing quirk that can segregate the interface quirk, but this would make the system too complicated and difficult to follow.
- The introduction of quirks and custom behaviours for each item can lead to increased complexity, potentially making the code harder to manage as more items and quirks are added.
- With multiple quirks and probabilities, the code may become error-prone, requiring thorough testing to ensure correct behaviour.

Design Choice #2

We can opt for a single class called IsolatedTraveller that combines all the responsibilities, including rendering the character, managing transactions, and handling quirks. We also have a transaction action instead of the two separate actions Purchase Action and Sell Action. The Isolated Traveller then processes these item prices within the class itself and then passes it on to the transaction action to perform actions on the actor that is trading with the merchant.

Pros

- This approach would be easier to follow and understand, making the implementation process more efficient with a small learning curve.

Cons

- This violates the **SRP**, as the class becomes too complex and takes on multiple responsibilities. Any changes or additions to merchant behaviour would require modifications to this monolithic class, making it harder to maintain and extend.
- Adding new items, quirks, or behaviours to the merchant is challenging and may lead to code duplication. Extending the merchant's inventory or introducing new quirks would likely require changes directly within the IsolatedTraveller class. This violates the **OCP**, as the class is not open for extension without modification.

Decision

Design Choice #1 adheres to most principles of SOLID. It ensures that each class has a single responsibility by creating a clean separation between the Actor, merchant logic, and quirks. This separation makes the codebase more modular and easier to maintain and extend over time. It also aligns with the **Open/Closed Principl**e by allowing for easy extension without modification. New items, quirks, or merchant behaviours can be added independently within their respective classes. This separation of concerns ensures that extending the merchant's inventory or introducing new quirks can be achieved without altering existing code. Instead of relying on a concrete IsolatedTraveller class,we depend on an abstract Actors abstract class. This decoupling promotes flexibility and allows for the introduction of various merchant characters with different behaviours by implementing the Actors abstract class without modifying the existing codebase (**DIP**). Since the other alternative was not following SOLID principles, we considered Design #1 more viable during the design process.

REQ 4

Design Choice #1

The feature to add the GiantHammer weapon to the game is identical to implementing weapons in assignments 1 and previous sections of assignment 2, where the GiantHammer class inherits from the WeaponItem class to enables it to implement the features of a weapon

The new design decision is for the GiantHammer to have a skill and be sellable. To achieve this it implements both WeaponSkill and Sellable interfaces. The former allows GiantHammer to have a skill and the means to turn off the skill, whilst the latter allows the GiantHammer to be sellable by being able to provide its price

Pros

- The use of inheritance in having GiantHammer extend from WeaponItem is a clear example of reusability and adheres to the concept of **DRY (Don't repeat yourself)**. In addition, as GiantHammer can now access functionalities of WeaponItem, it also implicitly adheres to the **Liskov substitution principle**. Finally, it adheres to the principle of **dependency inversion**, as its ability to inherit from WeaponItem allows it to be used by high level modules by depending on WeaponItem instead of depending on low level modules such as GiantHammer directly
- The use of interfaces (WeaponSkill and Sellable) ensures extensibility as they can be used for subsequent classes. This adheres to the **open-closed principle** as all subsequent classes that implement these interfaces can have different codes in their inherited methods without changing the way the methods are used
- The notion of implementing two interfaces not only circumvents the java feature of non-multiple inheritance, but also adheres to the principle of **interface segregation** as well as **single responsibility principle**. The former prevents interface pollution by ensuring that each interface are utilised to its fullest, with none of the methods inherited having no functionality, whilst the latter enforces that each interface is specialised in one functionality and nothing more (**Separation of constants**), is cohesive in functionality and contains everything to support that functionality.

Cons

- All subsequent classes that are similar to GiantHammer will have to extend / implement from the same classes / dependencies, incurring some repetition which violates the concept of **DRY (Don't repeat yourself)**
- The number of extensions from GiantHammer (Inheritance, implementing interfaces, etc) increases coupling and the overall complexity of the system.

Decision

Overall, the decision to adopt this approach to create GiantHammer and all its functionalities is satisfactory as it maximises reusability by building off from the pre-existing work from assignment 1 and previous sections of assignment 2. Since no other alternatives were considered more viable during the design process, we are happy to use this approach

REQ 5

There were 3 designs we considered with regards to the implementation of the Weather system feature. All 3 of the designs shared a common, invariant motivator behind them; that is to make Weather a globally accessible and enforceable feature.

Common Design Choice

The implementation of the Gate appearing at the ForestWatcher's death and the printing of a custom message upon Death is done by **dependency injection** from Application [new ForestWatcher(ancientWoodsGateObject)] that adheres to **LSP** by turning into a ground.

And calling the static variable storing a death message (FancyMessage.java) within an **overridden** unconscious method respectively.

Design Choice #1

Initially, we implemented Weather as a singleton that controlled the different weathers through an attribute of type WeatherTypes (a public enumeration) with a WeatherSusceptible (interface) that RedWolf, ForestKeeper, Bush and Hut would implement forcing them to define rainyWeather() and sunnyWeather() which would be the class specific changes as per Weather, promoting **Single Responsibility Principle.** The interface also had two default functions that checked if the WeatherType argument passed is the current weather globally to make sure only the right function (sunnyWeather() or rainyWeather()) would run, and forceWeatherChanges() a function that would force execute sunnyWeather() and rainyWeather().

Pros

- Easy to understand and high modularity: Using Enums to represent the states/types of weather and having an attribute in the Weather manager class makes it easy to follow. Moreover, each part of the functionality is modularized including the WeatherType equality check.

- **Single Responsibility Principle**: The WeatherSusceptible forces all interface contract methods (rainyWeather() and sunnyWeather() to be called every playTurn/tick) the delegation of whether or not that function is supposed to be executed at that stage is left to the function itself. This is obviously on top of the logic itself being delegated.

Cons

- Violation of the **DRY Principle** as each WeatherSusceptible Entity/Ground's constructor would have a forceWeatherChanges() call in it.

- This design also violates **Open-Closed Principle** as each new Weather state would have to have an if condition to check if it is the appropriate Weather currently to execute its contents, moreover, when adding new Weather Susceptible implementing classes, we would need to add a call to forceWeatherChanges() in their tick/playTurn functions which decreases extensibility.

- This would also have violated the **Interface Segregation Principle** as not all WeatherSusceptibles reacted to **both** Sunny and Rainy weathers.

Design Choice #2

Our second design, implemented a radically different approach and design. We recognized that having a default method of the interface that performed and equality check was a code smell and so we needed to shift to a state pattern based implementation, where a new Interface is created called WeatherState that has two concrete implementations called RainyState and SunnyState that is toggled between by the Weather singleton. This gets rid of the enum based implementation and further abstracts the execution of the sunnyWeather() and rainyWeather() by delegating to the respective state itself, increasing **Single Responsibility.**

Pros

- Code Smell of manually equating and checking for WeatherType is removed and Interface abstraction is used to delegate and modularize the flow of execution.

- Enums are removed and replaced by concrete implementations of the WeatherState interface making the code adhere to **Open-Closed Principle**

Cons

- Each WeatherSusceptible has now a processWeather() function that gets the current weatherState and executes the internal logic of that state which executes the correct weather specific function of the WeatherSusceptible this processWeather() function also needs to be called in every tick() and playturn() which breaks the **DRY principle.**

- The **ISP principle listed in Design #1 has still not been fixed.**

Design Choice #3

In light of the above cons listed for both the designs, we decided to implement Weather in a **push based** manner. Where we isolated Weather singleton which followed the previous state pattern and created a WeatherSusceptibleManager **singleton** to handle all RainySusceptibles and SunnySusceptibles, this is done by **registering** each RainySusceptible and SunnySusceptible to the respective list in WeatherSusceptiblesManager and then for each playTurn of the boss, calling the WeatherState's processWeatherState() **centrally and passing a singleton instance of**

WeatherSusceptibleManager to it which would execute all of either RainySusceptibles or SunnySusceptibles by calling their rainyWeather() or sunnyWeather() respectively.

Pros

- By making WeatherSusceptibleManager a **singleton** we follow object oriented programming principles and allow testing frameworks to easily test dummy data by leaving scope for **dependency injection** from ForestWatcher (pass a singleton instance as an argument) This also allows us to, in principle, integrate server support and facilitate HTTP POST requests.

- By **centralising** the call to processWeather() from the ForestWatcher's play turn, we do away with the need to have a call to the function mentioned above in each WeatherSusceptible's playTurn() or tick(), fixing the **DRY principle** violation.

- A push based implementation like this also ensures there is no equality checking between the current global weatherState and the needed weatherState to execute the WeatherSusceptible specific rainyWeather() and sunnyWeather() respectively, fixing **code smells.**

- By separating the WeatherSusceptible interface into a SunnySusceptible and RainySusceptible interface we allow and acknowledge the existence of classes like ForestKeeper that react to only certain states of weathers and not others, this fixes the previously mentioned **Interface Segregation Principle** violation.

- Code is highly modularized and follows **Open-Closed Principle**, the 2 interfaces abstract the implementation of Weather and States enough to make it very easy to add a new state in the future, for example, say a SnowyState.

Cons

- Code Comprehensibility suffers as a consequence of abstraction and increasing number of classes, one could revert back to a enum based representation of all WeatherTypes and have a case-switch to control weather specific changes in WeatherSusceptibles but that would trade of extensibility for understandability.

- Increasing Dependencies and Circular dependencies are introduced as SusceptiblesManager has two lists that objects **register** to using either the RainySusceptible or SunnySusceptibles default function. However, when the processWeather() is called it has a local dependency to both SunnySusceptibles and RainySusceptibles. However, because one of the classes is called before the other there is no conflict at compile or runtime. It is also worth noting that the engine itself has a similar circular dependency between Exit and Location.

Decision

We chose to implement the 3rd design as it is a quintessential example of a singleton/StateManager implementation that adheres to SOLID principles. This design also has the least amount of cons and is highly extensible with regards to both the processing of Weather and adding new Weather States themselves.

We have decided to allow registrations to use the getWeatherSusceptiblesManager() for flexibility right now as there is only one instance of the WeatherSusceptibleManager and injecting it as a dependency would create unnecessary dependencies as far as this project/assignment is concerned. However, we understand the need to **loosely couple** the WeatherSusceptibleManager and the WeatherStates and so we have demonstrated the **dependency injection and loosely-coupled** avenue by doing so in ForestKeeper.