1. Environments & enemies

A. Environments

Since the environments differ from grounds in that they are able to spawn enemies, a new abstract base class `Environment` that extends `Ground` is created to facilitate for the spawning of enemies in locations. It reduces code repetition by defining the environments' behaviour at each turn, which is to spawn enemies, by overriding `Ground`'s `tick` method. Then, each environment can implement its own spawning function, depending on the type of enemy that ground is meant to spawn. Because `Environment` extends `Ground`, it poses no issues in the game engine, as it is interacted with like any other ground (LSP). New enemy-spawning environments can be created easily without need to change base code (OCP). This base class is also useful when the enemy factory in Req 5 is implemented.

B. Enemies

Since all enemies share common functionality, an abstract base class `Enemy`, extending `Actor`, is created to reduce code repetition and improve maintainability for all enemies. Functionality like having multiple behaviours (with different priorities), holding weapons, and `allowableActions` are the same among enemies. Therefore, having a base class helps reduce repetition (DRY).

To handle the probabilistic despawning of enemies at every turn, a new `Behaviour` is realised with `DespawnBehaviour`. It determines whether `DespawnAction` (an class generalising `Action`) should be called. Although it might seem that implementing a behaviour for such a simple action is overkill, but it follows the `Enemy` class's use of prioritised behaviours in the `playTurn` function. In addition, it is necessary to store it as a `Behaviour`, because Pile of Bones does not get despawned in this way. Therefore, if a call to the `DespawnAction` is done in `playTurn` instead, it would require a complete override of the `playTurn` function, which would cause code repetition.

A flexible capability is created for each enemy type – Skeletal, Canine and Crustacean – so they can be identified when attempting to attack each other. The flexible status enables more enemies to be added in the future without modifications to existing code to stop enemies of the same type/group attacking each other (OCP). New types of enemies can also be added in a similar way.

To let enemies attack, a new `Behaviour`, `AttackBehaviour`, is created. It checks the surroundings for attackable actors, and returns an `AttackAction` if so. For special attacks, including enemy-specific skills and weapon skills, another `Behaviour`, `SpecialAttackBehaviour`, is created. This gets called by `AttackBehaviour` to

determine whether the attacking actor is able to do any sort of special attack, by checking the weapons for skills and character-specific attacks.

The different enemies in the game are implemented using `Enemy` as the base class. Pile of Bones also extends `Enemy`, however this implementation comes with some downsides. Pile of Bones' behaviour is quite different to the other enemies (it cannot move, cannot attack, cannot spawn, etc.), so it could be argued that LSP is sort of violated here, and its constructor overrides much of the base class's constructor. Despite these downsides, it is still worth using the `Enemy` base class, because the Pile of Bones still uses much of the functionality in the base class, including `playTurn`'s behaviour checking, `allowableActions`, and reset behaviour. Making it a normal `Actor` object would introduce many code repetitions.

The transformation of HSSs into Pile of Bones is handled using a new `SwapActorAction` (that extends `Action`). This action essentially replaces one actor with another on the game map. A new action has been made because it is more flexible, and can be used for other future functionality – this sort of action can be used for features like evolution of characters, or if there are other "un-dead" actors in the future. Un-dead actors, like Pile of Bones, "remember" their former actor, so when it comes time to revive, it will be able to call `SwapActorAction` again. Whether the Pile of Bones revives is decided by a `Behaviour`, `SwapActorBehaviour`, which essentially decides whether enough turns have passed to call `SwapActorAction`. This behaviour has been created to follow the game engine's use of `Behaviour`s and allow for a more flexible implementation, as well as be called by the `Enemy` base class's `playTurn` function (so the Pile of Bones does not have to override that function).

To implement character-specific skills, like the slam AOE attack in the Giant Crab, an enumeration has been created.

But, first, an abstract base class `AreaAttackAction` has been created to handle all AOE-type attacks, as they essentially do the same thing (get the 8 surrounding locations to attack). Other AOE-type attacks can extend this base class to reduce code repetition (DRY). The base `AreaAttackAction` class is abstract mainly because the `menuDescription`s need to be different so that the player can choose them correctly in the menu. The downside of this is that the classes only override the `menuDescription` function, and it seems a little over-complex for such basic requirements.

An enumeration, `SpecialAttackType`, is made to define the different types of character special attacks (like slam AOE attack). This allows it to be added to the enemy's (or actor's) capability set, and its corresponding special attack action can be obtained easily without `if` or `switch` statements (good for following OCP).

C. Weapons

As the Grossmesser is a new weapon item, a new child class of `WeaponItem` is created – `Grossmesser`. To address the spinning attack, a new `SpinningAttackAction` class extending `AreaAttackAction` based class is created, and is returned via the weapon's `getSkill` method. This is needed to be able to show to the player in the menu the option to use the weapon skill.

2. Traders & Runes

Currency management system

To implement the currency-related functionality in the game, a currency-management system has been designed to facilitate for these functions.

A class `RuneManager` is used to keep track of the Runes that the player holds. It has been made as a singleton to allow classes throughout the game to be able to access currency-related methods.

The singleton pattern means that one instance of the manager with its record-keeping data structure can be accessed by any other class as needed, so the records are always up-to-date.

However, a singleton pattern comes with its downsides. Having a single class handle all the currency-related functionality violates SRP, as the class has multiple responsibilities – checking balance, adding balance, deducting balance, resetting a player, etc. It is also difficult to control access to, because it can be accessed globally by any class (since it has a static instance-returning method).

Despite the downsides, the singleton pattern appears to be the most appropriate option for managing currency for now. The game system is not too complex, and being able to maintain one copy of the currency record that is up-to-date is quite an important requirement for this system.

Originally, it was intended for the currency-management system to handle multiple currencies for multiple players. Therefore, a `CurrencyItem` abstract base class (that extends `Item`) was made to allow other currencies to exist in the game. This base class would allow the currency manager to accept any sort of currency as the same (LSP). The single currency for now, `Rune`, would extend the `CurrencyItem` class and have its name and display character defined. But, for the sake of simplicity, only one currency can be held per actor for now.

A downside of this approach is that items of other types cannot be used as currency when trading, as they cannot extend `CurrencyItem` if they already extend another base class. An interface using DIP might be a way around this. As a sort of experimental implementation, a new `BuyingCurrency` interface is made (I don't know what exactly to name this). The interface is implemented by objects that can be used to buy items from the trader – it might be Runes or other items. This approach allows

trading to happen, because the `BuyTradeAction` (mentioned later) does not necessarily call `RuneManager` when an actor buys an item – it depends on what is being used to buy an item. If an item is bought using Runes, the `Rune` object's implementation of the interface's `deduct` function calls `RuneManager` to try to deduct the money. If an item is being "bought"/traded using a non-currency item, say, some sacred remembrance token (that I just made up), the remembrance token will also implement the `deduct` function, but it would instead just remove it from the actor's item inventory. It allows for a more flexible approach to payment.

Because `CurrencyItem` extends `Item`, it can be interacted with like an item, such as picking it up or having it lay on the ground (LSP). This is useful for when the player gets reset and needs to recover their runes.

To ensure that Runes can be recovered correctly, its pick-up action will be overridden to execute an alternative action, `PickUpCurrencyAction` that extends `Action`. Instead of adding the Runes to the actor's inventory, it will call the `RuneManager` to update the balance of the actor who picked up the Runes.

A. Enemies

To allow enemies to drop a certain amount of currency upon defeat by a player, a new interface `CurrencySource` can be implemented by the base `Enemy` class. Enemies that offer a reward (which is all at the moment) will implement the interface's function to return a certain amount of reward. This reward, if implemented by the child class, will be added to the enemy's item inventory (since the Runes can be treated as an `Item`).

Ideally, the function to generate the reward as part of `CurrencySource` would be called directly by the `DeathAction` to transfer Runes to the player attacker. However, a way to achieve this without downcasting has not been found. Though, it would provide a much simpler and "cleaner" implementation. Because the base class `Actor` cannot be modified, there is no way for the `DeathAction`'s `execute` function to call the reward-generating function. Therefore, a downcast to the `CurrencySource` interface would be done (via an `if`-statement), before calling the reward-generating function.

As a workaround to avoid downcasting (because apparently downcasting is always bad), albeit more complicated, two new statuses have been created. One states that the status holder can only be dropped upon defeat by a player, and another states it is to be immediately picked up by the attacker upon target defeat. These statuses are applied to a `Rune` item that is added to the `Enemy`'s inventory at instantiation. This allows the `DeathAction` to identify the Runes (though not specifically identify it as a `Rune` – no downcasting here) stored in the enemy inventory and deal with it appropriately (only drop when attacked by a player and to immediately call the Runes' pick-up action instead of dropping it). This is admittedly not an elegant solution, but it

avoids using downcasting, and is technically more flexible, as other items in the future could potentially require what is effectively a transfer action, or items that, as well, should only be dropped when attacked by a player. (Player checking is done by `HOSTILE_TO_ENEMY` status.)

B. Trader K

As the trader is effectively introducing another actor of special abilities, a new abstract base class `Trader` is created that extends `Actor`, which allows it to be interacted with by the game engine and, hence, the player. It can provide options through overriding `allowableActions` to present the trading options available from the inventory of buyable and sellable items. To ensure Kale cannot move around, the `playTurn` function simply returns the `DoNothingAction`.

This implementation sort of violates LSP, as the trader is quite different to how other actors in the game behave (does not really use `playTurn`, does not have any hit points or other attack characteristics). However, it is necessary to have it extend `Actor` so that Kale (and traders in general) can exist on the map and be interacted with by the player.

C. Weapons

To facilitate buying and selling items, two new interfaces are created – `Buyable` and `Sellable`. Objects that can be bought or sold implement these interfaces respectively. This allows, essentially, any object to be exchanged via the trader. The interface can also be used to define the buy and sell prices, so that they do not need to be handled by the trader. Although, this does mean that the price is the same across all traders (though that is not an issue for the requirement). Using DIP allows trading to be more flexible in terms of what can be traded. ISP is followed so that an item only needs to implement whether it is buyable or sellable, and not both, if they cannot be exchanged both ways. Using an interface also enables these items to be given or taken from the actor differently depending on the item. For example, a `WeaponItem` can only be given to a player using the `addWeaponToInventory` function, and not the `addItemToInventory` function. Having the buyable items implement the `Buyable` interface, for example, allows a buyable weapon to call `addWeaponToInventory` and a buyable standard item to call `addItemToInventory`, eliminating the need for any sort of downcasting when giving or taking items from/to an actor.

One small downside of this design (though still follows requirements), is that a specific weapon cannot be specifically sold, meaning if there are two weapons of the same type (say, two Grossmessers), selecting to sell one will just remove either one from the player's inventory. This is not currently an issue, but could become an issue if weapons hold attributes that can change (say, the number of uses). Implementing such improved behaviour would also need changing how the item is presented to the

player, as well as changing how items are selected and removed from the player's inventory.

To allow the player to select a trade action in the menu, two new `Action`s are created – `BuyTradeAction` and `SellTradeAction`. Both are similar in that they take an item to be bought or sold, then calls the required methods to execute the trade. These actions can be executed via the game engine's action-execution logic.

3. Grace & Game Reset

A.Flask of Crimson Tears

A new class for `FlaskofCrimsonTears` is added which extends `Item` as it has similar characteristics such as being portable (carried by the player) (LSP). This class implements `Resettable` as the number of uses is refreshed on game reset, and `Consumable` which is another added interface. A new action `ConsumeAction` extending `Action` is also added to handle the operations performed by the consuming of items.

Instead of letting the `FlaskOfCrimsonTears` handle the operations, this new interface and action are added so that new consumable items may be added to the game with minimum modification (OCP).

B. Site of Lost Grace

`SiteOfLostGrace` is a class added to the game and extends `Ground`. A new action, `RestAction` extending `Action` is also created and this class calls reset when executed. To let the player interact with the site this class adds `RestAction` to the allowable actions list, and to let only the player interact with the site, `Status.HOSTILE_TO_ENEMY` is utilised. As of now, there is only one site of lost grace which is located within the floor which the enemies cannot access, and therefore, there is no need to use `Status.HOSTILE_TO_ENEMY`. However, this condition check is added to ensure that if other sites are added to the game later, enemies may not interact with them.

Creating a new action which allows the player to rest also ensures that even though unlikely, sites or locations similar to sites of lost grace can be added to the game later.

C. Game Reset

As there are two ways to reset the game either by resting at `SiteofLostGrace` or by player death, two enums of `ResetType`, `RESET_ON_REST` and `RESET_ON_DEATH` are created and when reset is called, it checks the reset type and appropriate methods are performed.

By defining two types of reset, classes can behave differently depending on the type of reset, such as items that might not be removed upon a certain type of reset. This is useful for extending the game in the future, where more reset behaviours might want to be defined.

A new class `ResetManger` is added and it acts as a singleton to handle the reset operations.

Singleton method is used as it was the most suitable method for our implementation of the game and as it did not have many operations to perform in the first place. `ResetManger` when instantiated creates an array and adds items and enemies that implements `Resettable` to the list.

Even though using a singleton class breaks SRP, in this case the resetting follows the Observer behaviour where multiple objects subscribed to the manager class can be updated at once. In this case `ResetManager` acts as the class being observed or being subscribed and the instances of other classes implementing `Resettable` can be regarded as subscriber classes. This observer behaviour is used so that when reset is called, all classes implementing `Resettable` are updated or reset.

Furthermore, having a singleton class fortifies the idea of having only one record of the resettable items.

Actions performed when reset is called are defined within the abstract classes implementing `Resettable` such as `Enemy` and `CurrencyItem` instead of being defined within individual classes to eliminate the repetition of code (DRY).

As new items and enemies which can be reset can be added to the game without having to change the other resettable classes or `ResetManager` it follows OCP.

Initially, to reset the game, a new class `ResettableGameMap` was to be added so that the locations of the items can be accessed but this is repeating the code in the engine. Instead an alternative method to pass the game map as a parameter called `DespawnAction` is used, which has much better maintainability, and better conforms to the requirements. This class despawns or removes the enemy from the map when called and is also used in Req 1 and 5.

4. Classes

It was considered to create an abstract base class for Combat Archetypes, and have the child class extend the base class. These classes would be added to a list-type data structure in the `Application` class to display to the player. However, a method using enumerations was found that arguably has better extensibility. By defining the archetype classes using an enumeration, it ensures that there is only one instance of the archetypes. An enumeration also has easier extensibility, as an entry to the enum definition is all that is needed to add a new archetype (OCP), compared to having to

create a new child class and adding it to the `Application` class using the abstract-class alternative. And by `Application` class, player can decide which combat archetypes they would like to choose.

`UnsheatheAttackAction` and `QuickstepAction` are two special attack actions. By defining these two special attack actions as dependencies of the `AttackAction` class, it reuses the standard `AttackAction`, which helps with reducing repetition of the attack logic. It allows the behaviour and functionality of each special attack operation to be encapsulated in its own class, improving maintainability. These two special attack actions an also be considered generalisations of the Action class because they share many of the same attributes and behaviours as other actions. This also ensures that the game engine is followed. They will share common functionality, such as executing an action and checking for prerequisites.

To account for the modified weapon characteristics of the `UnsheatheAttackAction`, a new `ModifiedWeaponItem` class is created using an existing `WeaponItem` to be "cloned" but with modified attack stats (damage, hit rate). It is not the most elegant solution, as it seems to try to work around the abstract base class, though it does require an existing `WeaponItem` in its constructor to ensure that it is not mocking the base class. But, since all weapon characteristics are private and immutable, it appears to be the only way. This would allow the modified weapon to be given to the `AttackAction` to be used, and it would not affect dropping the weapon, because the original weapon in the `Actor`'s inventory is unaffected. An alternative would be to create a special modification of the `Uchigatana` called like `UnsheatheUchigatana`, but that is not extendable in case other weapons also need to have their stats modified in some other action. Yet another alternative could be to modify `AttackAction` to take some parameter of modified weapon stats, but that seems to be even more over-complex.

Two weapons `Uchigatana` and `GreatKnife` will extend the `WeaponItem` class so that it can be used like other weapons in the game by the player and enemies (LSP). For the Club, which does not have any special skills, it is appropriate to extend from the WeaponItem abstract class.

5. More enemies

East–west game map

The east and west sides of the game map are implemented using an abstract-factory pattern. An interface, `EnemyFactory`, provides the methods of the possible types of enemies to be spawned (since the differences in the sides of the map are to do with the different enemies being spawned of one type). Two concrete classes, `WestEnemyFactory` and `EastEnemyFactory` implement the `EnemyFactory` methods. These classes provide the side-specific enemies to be spawned. Utilising the abstract-factory pattern allows the `Environment` base class to accept any sort of

`EnemyFactory`, utilising polymorphism to allow new sides of the map to be added later without modifications to the base class (OCP). Each `Environment`-extending ground will implement an abstract function in the base class to decide which type of enemy to be spawned from the given `EnemyFactory`.

New sides of the map can be easily created by defining a new concrete implementation of the `EnemyFactory` interface, then passing it to the `Environment`. No other classes require modification.

Although this design works well, it does come with its downsides.

Firstly, because the types of enemies to be spawned are defined as functions in the base interface, adding new types of enemies would require a new method in the interface, as well as implementing it in all the `EnemyFactory` concrete classes. This violates OCP, but its trade-off is a system that is simple to extend with new sides of the map and new enemies.

Secondly, the `Environment`s cannot detect their side of the map. This is not an issue for the current requirements, though it means that it is possible that an error is made where an `Environment` whose `Location` is on the east side gets passed a `WestEnemyFactory`, for example, as it is not known by the compiler (unless assertions are added). Although the side of the map can be passed to the `Environment`s via, say, an enumeration, it would mean that a `switch`-statement is necessary for the `Environment` to check which `EnemyFactory` to use, which hinders OCP. And, the current system does not require any other side-specific features to be added. If this becomes a requirement later, the aforementioned solution could be useful if its added benefits help outweigh its downsides.

A. Enemies

The new enemies are similar to the ones introduced in Req 1, so it will not be explained in detail here.

The flexible status to indicate enemy type introduced in Req 1 becomes useful in this requirement, as the new enemies all fall under some enemy type (skeletal, crustacean, canine), and it prevents enemies of the same type attacking each other.

Other requirements are handled by the `Enemy` base class (also from Req 1).

B. Weapons

The Scimitar is a weapon similar to the Grossmesser (with Spinning Attack as a special skill), so is implemented in a similar way to Req 1. The weapon will be added to Merchant Kale's inventory of buyable and sellable weapons.