FIT2099 ASSIGNMENT 3 : DESIGN RATIONALE
GROUP 3: Carter Hills, Devshi Janakantha
DESIGN RATIONALE

Presented is the design rationale accompanying the UML class diagrams and sequence diagrams associated with the 'Design o' Souls' proposal.

SOLID principles are the forefront of the design rationale, but a summary is provided here to avoid repetition. As this game largely extends on the preexisting engine, 'single responsibility' and 'open to extension, but closed to modification' principles are the primary ones used. Examples of these include separate classes for most game entities, and extending on the game engine code wherever possible. Also, grouping similar functionality into one, for example in the rest/reset and fogdoor/bonfire transport functionality mean the code is only required to be written once.Other principles, such as Dependency injection (through constructors) and design by contract are also heavily used, but are not as central as those aforementioned.

Ground Revisions:

GroundType abstract class added: in order to ensure all classes that extend this class are equipped with the common methods the game's entire ground will share, so code is not repeated. Additionally, because the Ground class cannot be adjusted. Methods specific to a class (eg. Cemetery would implement a method for spawning Undead) can hence be added to the class itself, allowing different ground types to implement different functionalities..

Created package (within Game package) named 'GroundType' that consists of the newly added GroundType abstract class and relevant Ground 'types'.
This is to reduce clutter within the game package, and also create a common location for all classes that extend the abstract class GroundType (Cemetery, Dirt, Floor, Valley and Bonfire) and to emphasise the distinct inheritance connection they share.

Cemetery class added, dependent on Undead class**:** A class that represents Cemetery grounds. This class is also able to generate at least one Undead, hence the Undead class is dependent on the Cemetery class.

Bonfire class added, which provides reset action to player: This class represents the Bonfire situated in the middle of the game map. This class should enable the player to perform a soft reset by returning the ResetAction to the player.

BonfireManager, organises all bonfires: Class that contains all bonfires within the game, which is imperative to the bonfire transportation functionality. Organises bonfires by setting the player's current bonfire, and also tracking active bonfires and their locations. This class is created at the high level, and passed to all bonfires and the player.

LightBonfireAction, extends Action, depends on Bonfire: This class enables the player to light a bonfire in order for it to be able to rest or travel between other bonfires. It is returned by an unlit bonfire.

FogDoor class, extends Ground:
This class represents the fog door, the ‘portal’ that the player can use to travel between maps. It will return a TransportActorAction to transport the player. Enemies cannot use this functionality nor step on the fog door, which is implemented through the use of an enum.

TransportActorAction, extends Action: Transports the player to a target location. Thai is invoked by both the fog door, and also the bonfire in order to transport the player between them. It extends action for the display and selection capabilities and is dependent on the Location class through the constructor, in order to have a target location.

Item Revisions:
EstusFlask class created: This class will extend ConsumableItem, as the EstusFlask should inherit actions such as accessing the current allowable actions and ConsumeItemAction (see ConsumeItemAction).

The Estus Flask shares a dependency with Player, as opposed to the Actor class. This was decided as only the main player holds a single Estus Flask, rather than all Actors (which include enemies).

The EstusFlask class also implements the Resettable interface. This is because during the soft reset the player’s Estus Flask is refilled to its maximum charge. Hence the Estus Flask must be registered as an instance that is ‘resettable’ (i.e implements Resettable interface) so that the ResetManager when called in whichever context (entering Bonfire or Death) will ensure the potion is refilled/reset to its maximum charge.

EstusFlask extends the ConsumableItem class to ensure only items that are consumable can access and execute the ConsumeItemAction (given that its charge is above 0).

CinderLordItem class created: The purpose of this class is to represent the ‘Cinders of a Lord Item’ that Yhorm the Giant (and other Lords of Cinder) drops when killed.

Hence this class extends Item, due to it needing to inherit methods regarding portability such as drop.

TokenOfSouls item created:
The Token of Souls is generated and dropped when the player is killed. This was easier to implement than extracting the item from the inventory, when it has no other functionality in the game.

It was decided that the TokenOfSouls should extend the class Item, as Actors store Items and the portable ability of the TokenOfSouls is implemented. Only the player can pick up this item.

The TokenOfSouls also implements the Soul interface, as it should have the ability to transfer souls when picked up by the player after the game’s reset, and also have souls transferred to it when the player dies.

PickUpSoulsAction class created:

An action that extends from and slightly modifies the behaviour of PickupItemAction, in that when a Souls item is picked up, it transfers the souls to the actor and removes the item.

Chest class added, dependent on Mimic enemy and TokenofSoulsItem: This class represents the Chest, which when opened can drop the token of souls or spawn a Mimic. It extends the Item class so that it is easily removed from the map.

ChestOpenAction, dependent on Chest class: This class allows the player to open a chest, which either spawns a mimic or drops tokens of souls, removing the chest after the action is completed. Hence, it is also dependent on the token of souls and the Mimic enemy. It extends the game engine's Action class to enable regular action functionality including menu display, etc.

Enemy Revisions:

Enemy abstract class created: This class is inherited by all types of Enemies (Undead, Skeleton and LordOfCinder). This is to avoid repeated code (DRY), as enemies share common tasks which will be expanded on in the following section:

This Enemy abstract class will also implement the interfaces: Soul, Behaviour and Resettable. This was decided as:

- The Soul interface ensures that all enemies are able to transfer Souls to a player upon death.
- The Behaviour interface allows enemies to carry out their following behaviour, a behaviour shared by all enemies.
- The Resettable interface ensures relevant instances of enemies are registered as Resettable, hence upon the player dying enemies will be either relocated or removed from the map.

Skeleton class added to Enemy package: this class represents a skeleton. As this type of object also shares the goal of following and attacking the player, and is also deemed as an enemy it hence should extend the abstract class Enemy.

Created Enemy package: This is to reduce clutter within the game package, and also create a common location for all classes that extend the abstract class Enemy (Skeleton, Undead and LordOfCinder) and to emphasise the common inheritance connection they share.

Mimic class added: This class represents the mimic enemy that is spawned when a chest is opened. It extends the enemy class in order to implement the attack functionality and behaviours outlined for other enemies.

Create YhormtheGiant class and LordofCinder Abstract class:

These two classes are responsible for the features unique to the Lords of Cinder, which at this stage is solely Yhorm the Giant. Hence, the functionality common to other Lords of Cinder in the future will be moved from the class YhormTheGiant to LordofCinder in the

future. Such functionality unique to these classes are the “ember mode” passive skill, disabling of the WanderBehaviour common to all other enemies.

AldrichTheDevourer class added: This class extends the LordOfCinder class, as this enemy implements many of the same behaviours as Yhrom the Giant, the other lord of cinder. Aldrich’s own class enables it to carry a unique weapon, and have a unique display character, amongst other small functionality.

ConsumeAbility Interface
Required for actors that are able to consume items. Contains maximum HP getters.

ConsumableItem
Implementing a Consumable interface was decided against so as to avoid downcasting, Instead items that are consumable will instead be constructed as ConsumableItems which extend the Item class. Thus only objects of this class have the ability to carry out the ConsumeItemAction (see next).

ConsumeItemAction
Class responsible for consuming an item. Will enable all Actors to consume items to regenerate their health, and can handle multiple different ConsumableItems should they be added to the game. Placed in the actions package as it is similar to other actions that the Player or Enemies can complete.

Contains an attribute ConsumableItem to ensure that only instances of ConsumableItem (and are hence consumable) are passed into this action.

AttackBehaviour
The Attack behaviour class enables enemies to automatically attack the player through the detection of another actor that is hostile to enemies via the checking of exits. If such an actor exists, the enemy will execute an attack. Note that a hostile to enemy enumeration dictates the ‘attackability’ of other actors. This action extends the FollowBehaviour class to avoid repeated code.

<u>Reset Functionality:</u>

Thus far in the game, ResetManager is responsible for handling all reset functionality, and can be instantiated in multiple ways, both passively and actively by the Player. Hence, the reset manager, which controls the methods associated with resets is called by either the Player or by an action that the Player can select.

ResetAction Class:

The reset action class is responsible for the implementation of all the reset functionality that occurs when the player dies or manually at the Bonfire. It is automatically called by the player when the player is unconscious and is returned as an available action when the

player stands on the bonfire. This class also determines the appropriate actions to perform depending on the player being alive or dead.

Bonfire Reset Functionality:

To enable the player to cause a soft reset to occur, the Bonfire ground piece will have an action *ResetAction* associated with it that becomes available to the player when in proximity to a Bonfire. This extension from the action item means that the pre-existing functionality associated with selecting and displaying Actions can also be implemented for this Action.

Player Death Soft Reset Functionality:

The soft reset can also be initiated automatically when the player dies, either by being killed by an enemy or by stepping on hostile terrain (i.e. a Valley). This is handled in the Player's turn, where before performing an action the loop checks if the player is dead, or if it is standing on a valley, in either case performing a reset.

The primary difference between the active and passive resets is the transfer of player souls to the token of souls, and the subsequent placing of it. The token of souls is placed by the Player before the reset is called, enabling all reset functionality at this stage to be implemented in one place. It should be noted that should extra functionality be required with future updates, it will be modified via the ResetManager or the RestAction class.

Initially, it was proposed that a Valley object would detect a player on it and then call the reset, however this contradicted the object hierarchy in the game and seemed much more complicated to implement and modify.

Weapon Functionality:

Creation of WeaponSkills interface:

Passive and active skills associated with weapons are all to be implemented through the use of ‘Action’ classes which define the unique skills able to be performed by weapons. All weapons in the game will require methods to manage their active and passive skills (returning null if none available), where skills are stored within the weapon instance, depending on its type. Multiple designs were deliberated to arrive at this, and this approach is the most modular, enabling any weapon to possess any skill – useful when adding more weapons in future revisions. Alternatives included methods inside classes or modification to the AttackAction class, however those were discarded due to lack of modularity.

ChargeAction and WindSlashAction Classes Created:

Classes responsible for skills StormRuler has access to, hence the association in the UML class diagram. These methods will perform the action, similar to AttackAction when an Actor wielding the weapon has a turn, in line with the game rule of charge-windslash actions.

ChargeAction is accessible upon a Player picking up the StormRuler. Due to this action only being carried out by the Storm Ruler in this current design, it is likely that functionalities added to this weapon class will be abstracted if expansion is required in the future.

WindSlashAction is available after the charge is complete, and only on Yhorm the Giant. It stuns the boss and deals double damage. It is only available as an attack on Yhorm the Giant.

BroadSword, StormRuler, YhormGreatMachete, DarkmoonLongbow Classes Created:

Individual classes created to implement each of the four weapons in the game. These extend from MeleeWeapon, which is to be modified to interface with the WeaponSkills interface (see WeaponSkills description). All weapons, along with the parent class MeleeWeapon are to be refactored into the *weapons* package for ease of navigability. Skeletons and the Player wield a Broadsword when instantiated. Note that due to lack of similarity between weapons at this time, there are no subclasses of MeleeWeapon, but with further expansion further organisation of weapons types through an abstract class may be required. Note that the longbow, as a ranged weapon, has its own functionality that is discerned by the AttackAction class.

NoWeapon Class created:

This class handles the functionality for the Undead and Mimic weapons, namely overriding the default display option, enabling the printing of “No Weapon” in an enemy description. It extends the IntrinsicWeapon class.