**FIT2099 Assignment 1 & 2 [Updated from Assignment 1]**

MA_Lab4Team1
Lionie Annabella Wijaya (31316115)
Yeoh Wai Liang (32241674)

## Note

We have removed and changed some parts of design rationale, class diagram, and interaction diagram required for group of 3 due to a member leaving the group.

## Design Rationale

### Player and Estus Flask

Application creates our Player instance, Unkindled, Player class extends Actor class to inherit attributes and methods as an actor in game. Displaying the player's HP, weapon, and soul in console is possible by overriding playTurn and toString methods inherited to include additional player's description.

As an actor, Unkindled can store items in inventory. Two items are added to the inventory, a Broadsword instance and an EstusFlask instance. Both instances are subclasses of Item thus allowing them to be created and added to inventory, storing inside inventory is preferable because inventory of each actor is traversed while the game runs and can provide actions. Items can be added either when Player is constructed or created in Application first before added after the Player is constructed. We selected the first option as it is simpler and will not be confused with creation of other items or components in Application.

EstusFlask class is created which extends abstract class Item instead of PortableItem as EstusFlask cannot be dropped. EstusFlask has a charge attribute, which can be checked to indicate if drink action is available. To enable future extension of possible drinks or other actors as consumers (Open-closed principle and Interface segregation principle) and reduce dependency (RED principle) of EstusFlask to an action, DrinkAction is created and extends Action, which accepts instances of Drinkable and Consumer. Any drinking action should provide these, therefore Player class implements Consumer interface and EstusFlask implements Drinkable interface. Using these interfaces also allows the Dependency Inversion and Injection principle to get max HP from a consumer to provide information needed by the drink method in Drinkable when DrinkAction is executed.

### Bonfire

[Optional] Reset the enemies' position, health, and skills /Implemented/

Application creates a single Bonfire instance, Firelink Shrine, positioned in the middle of the map. Bonfire class is created which extends abstract class Ground to enable display in the game map and provide interaction with Unkindled through ResetAction. To prevent other actors interacting with Bonfire, an additional check is added to provide ResetAction only for actors (Player) with rest ability. ResetAction has an attribute lastBonfireRested which is useful to select which Bonfire to spawn back Unkindled in the future, it is set to static to enforce all ResetAction to have the same Bonfire to be selected. ResetAction extends Action and provides two constructor, this is needed to be able to handle two situations, one to handle player resting and another to handle player soft reset when he dies (further explanation for this situation in Soft Reset/Dying in the game). The ResetAction is designed this way to enable reuse on all scenarios requiring reset and avoid repetition (DRY principle) of creating multiple actions, such that we only need to add a few checks and adjustments before running the reset manager.

When RestAction is executed to handle player resting, lastBonfireRested is updated before reset manager is run. Running reset manage singleton will reset all Resettable instances (Single-Responsibility principle) accordingly based on their requirement. Therefore, all these classes: Player, Enemy, and EstusFlask should implement Resettable and should be registered during construction and override resetInstance method. Player and EstusFlask reset will only overwrite HP and charge attributes, but resetting enemies on the other hand requires more work since there are two possible scenarios (enemy is removed or reset and repositioned at initial location created). To handle this, we separate the work into Enemy class which will handle enemies that are removed during reset and a subclass of it, ResettableEnemy class, implements Repositionable interface to allow reposition of enemies. Each enemy will extend from one of these two classes and in their playTurn method will return RemoveActorAction or RepositionActorAction when reset is called.

## Souls

All classes which relate to Soul including Player, Enemies, and TokenOfSoul should have some mechanisms to enable exchanging, subtracting, or adding souls which is achieved by implementing the provided Soul interface. The methods in each class are overridden, the Player class should be able to transfer, add, and subtract souls as he is the only one collecting and using the souls while the Enemy class should only be able to transfer souls and TokenOfSoul class can transfer and add souls. In addition to this, each class has a soul attribute to indicate the number of souls it holds, which is used as a parameter for soul related methods. To collect souls from enemies is only possible if the enemy dies from an attack, thus an additional check is added in AttackAction to transfer souls from enemy to player when it dies.

## Enemies

[OPTIONAL] Enemies cannot enter floor, cannot attack each other, can randomly use active skills from a weapon /Implemented/
[OPTIONAL] Undead has 10% chance of dying in each turn if not under attack or not following player /Not Implemented/
[OPTIONAL] Lord of Cinder drops Cinder of Lord when it dies /Not Implemented/

Enemy class is added in which extends the actor class to provide interactions with actors and capabilities. Undead extends Enemy class while Skeleton and LordOfCinder classes extend ResettableEnemy class. LordOfCinder class is set as an abstract class as Yhorm the Giant is not the only possible enemy of type Lord of Cinder, therefore lord enemies, YhormTheGiant class, extends LordOfCinder class. Alternatively, we can ignore creating Enemy class and directly extend Undead, LordOfCinder, and Skeleton classes to Actor class. We selected the first option to provide polymorphism to different enemies and use inheritance, overriding, abstraction, and interfaces to implement different requirements applying to different types of enemies. By default, Skeletons and Lord of Cinders are created by application at the start of the game and Undeads are spawned by cemeteries placed in the maps (more information about spawning in Terrains). To extra messages or modification when an enemy is dead from an attack, the attack action's execution can be modified. Note that non player actors will drop non portable items if they die, this can be used to implement a situation where Lord Of Cinder drops CinderOfLord which extends PortableItem when it's dead by adding it to inventory during construction. Additional information about HP and weapon hold in the console is done by overriding the toString method.

For enemies that wander around (Undead and Skeleton), WanderBehaviour is added to their behaviour during construction. To enable the following player, Enemy class' getAllowableActions method is overridden to add FollowingBehaviour if the other actor interacting with the enemy is a player, additionally the resetInstance in ResettableEnemy class will remove this behaviour when game is reset. To prevent other actors entering the floor, an additional check is added to allow actors (Player) with floor entry ability to enter. To enable attack action from enemy to player only, an additional check is added to allow actors (Player) with hostile to enemy status to be granted with an AttackAction, the same applies to enable attack action from player to enemy, an additional check is added to allow instance of enemy only to get AttackAction from player.

While processing enemies each turn, different enemies will have different customization inside the playTurn method to determine action taken. In general the enemies will return one of these consecutive orders, either an action associated with the reset, attack or weapon action, behaviour action, or do nothing action is returned. Applying Single Responsibility principle, playTurn strictly gets the actions by parameter or calling other methods after some checks to return the action. To enable random active AttackAction, a method is implemented to filter out all attack and weapon action and randomly returns one to be executed in playTurn if available (also Single Responsibility Principle). Depending on the enemy, additional checks can be added to set, perform things, or return an action can be added (e.g., checking stunned status, consciousness, checking if undead is removed on 10% chance, etc).

## Terrains (Valley and Cemetery)

Both valleys and cemeteries are areas in the map and they should be added in as classes which extend from the Ground class to provide display, capabilities, and interactions with actors. Application should create several cemeteries. Valley class overrides methods to guarantee only actors with fall ability (player) can step on them and reduces player's HP to zero.

While the game is running, in each process of the player's turn, cemeteries may randomly spawn an undead. A method can be created separately (Single-responsibility principle) inside the Cemetery class that creates and returns an Undead object or null by the given probability, and this method is invoked for each cemetery in each turn inside the World class. If successful and return is not null, the World class will add the Undead inside the game map. Alternatively, we can calculate probability beforehand in the World class for each cemetery to determine if an Undead is created and construct an Undead by calling a method in Cemetery that always returns an Undead instance if the boolean value is true. We decided to go with the first option as it is clearer and if in the future refactoring is needed such as if the criteria of spawning Undead changes, we only need to refactor the spawn method itself.

Soft reset/Dying in the game

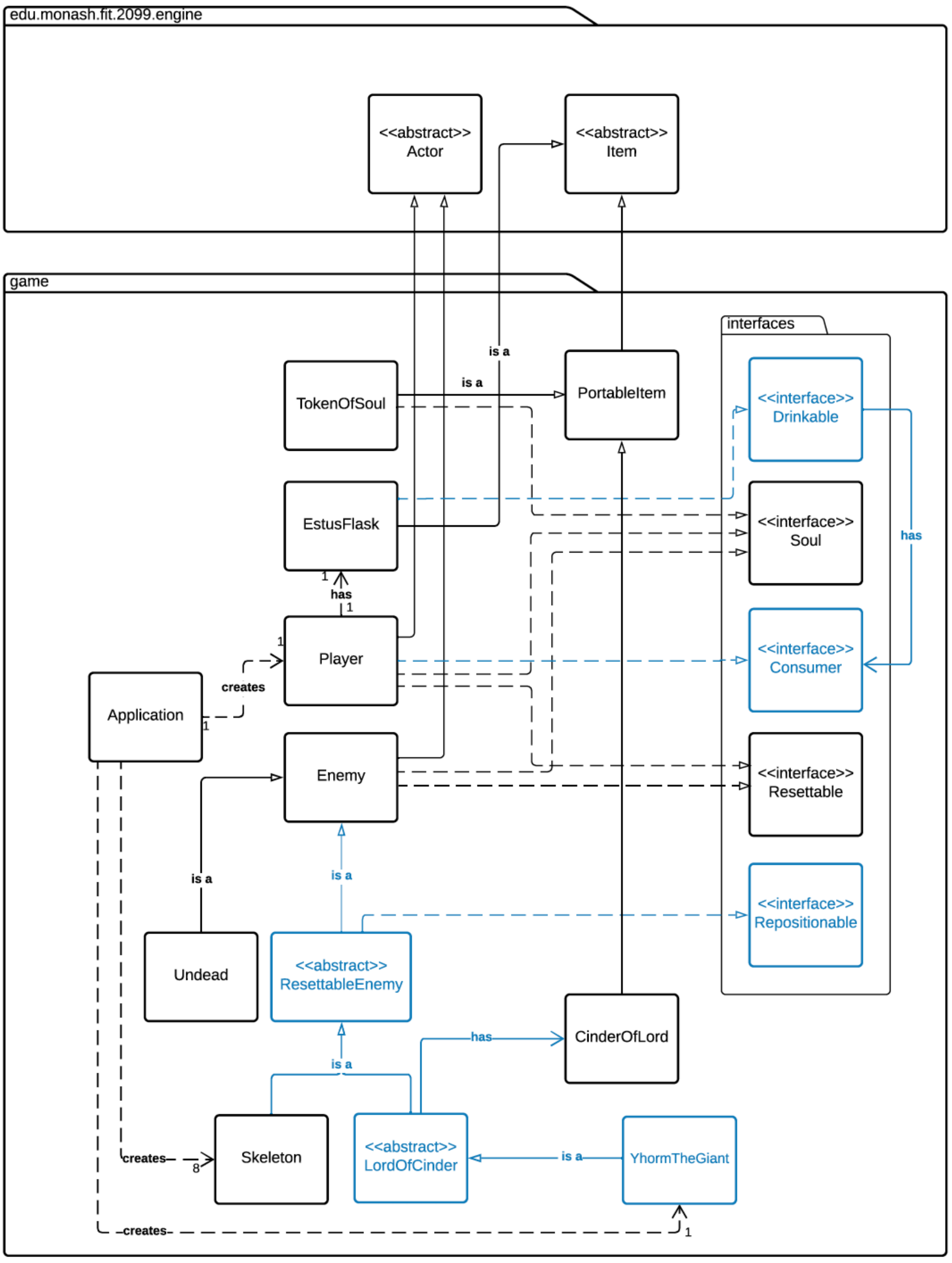
When the player dies then everything in the game will be reset and the player will respawn at the latest bonfire. We can reuse the implementation of the resetAction (DRY principle) described earlier in Bonfire. Prior to running reset manager, we need to put Token of Souls at the last location and spawn the player at last bonfire rested. Both information can be received from the player's attribute and map method to determine location. To solve the problem when a player dies in a valley, an enum Direction is created separately to handle things related with direction (Single Responsibility Principle), including calculating the latest location before falling from the hotkey selected and the valley's location. Afterwards, the Player creates an instance of TokenOfSoul and transfers his number of souls to the token before adding the token at their last dying location. TokenOfSoul extends PortableItem to enable pick up action later in the game. If the player picks up the token, the token adds it's number of souls to the player and gets removed from the map - which we can achieve by creating a PickUpToken action which extends PickUpAction and overrides getPickUpAction method to return PickUpToken action .

Weapon

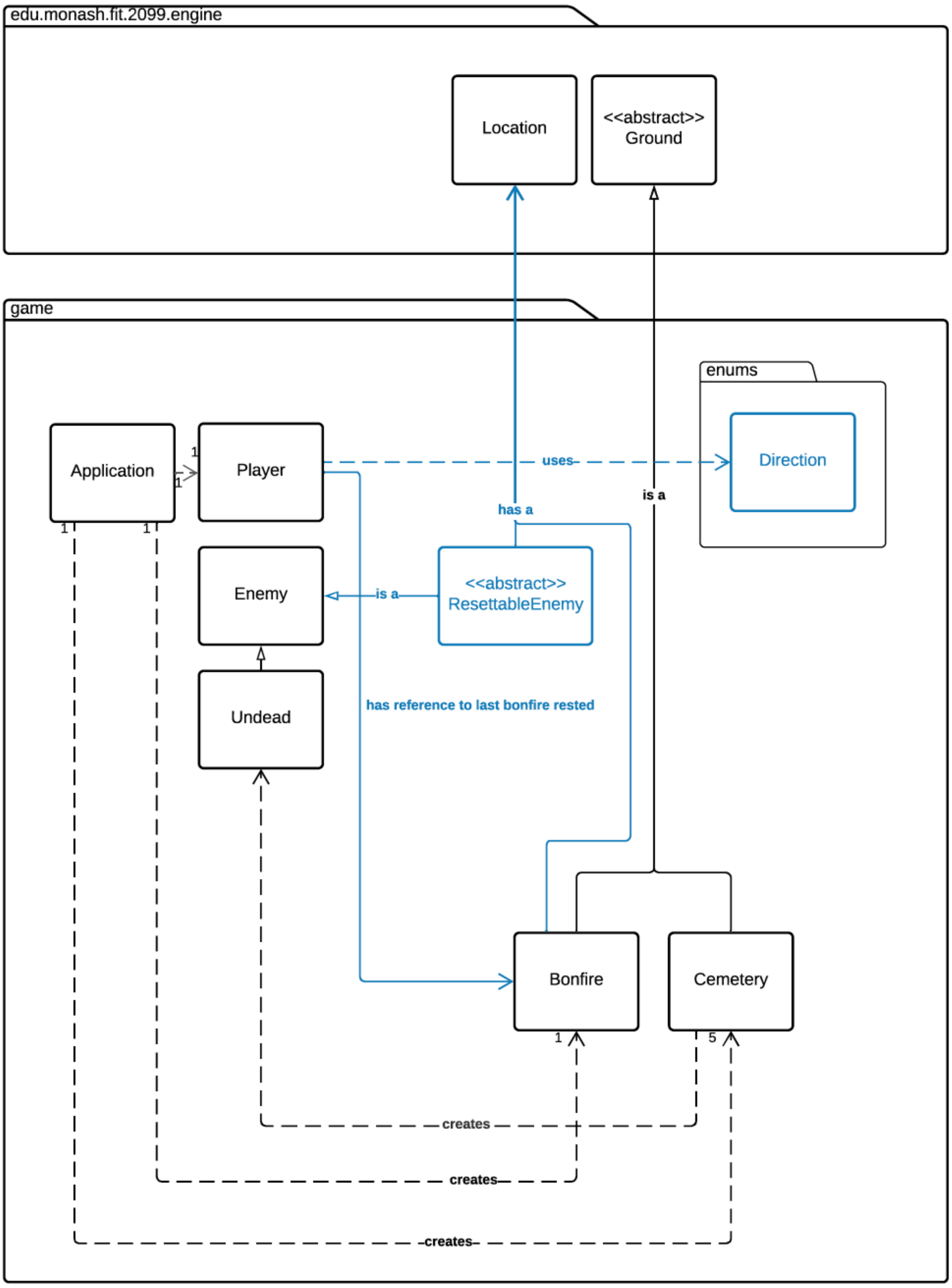
[OPTIONAL] Dullness passive skill for Storm Ruler /Not Implemented/

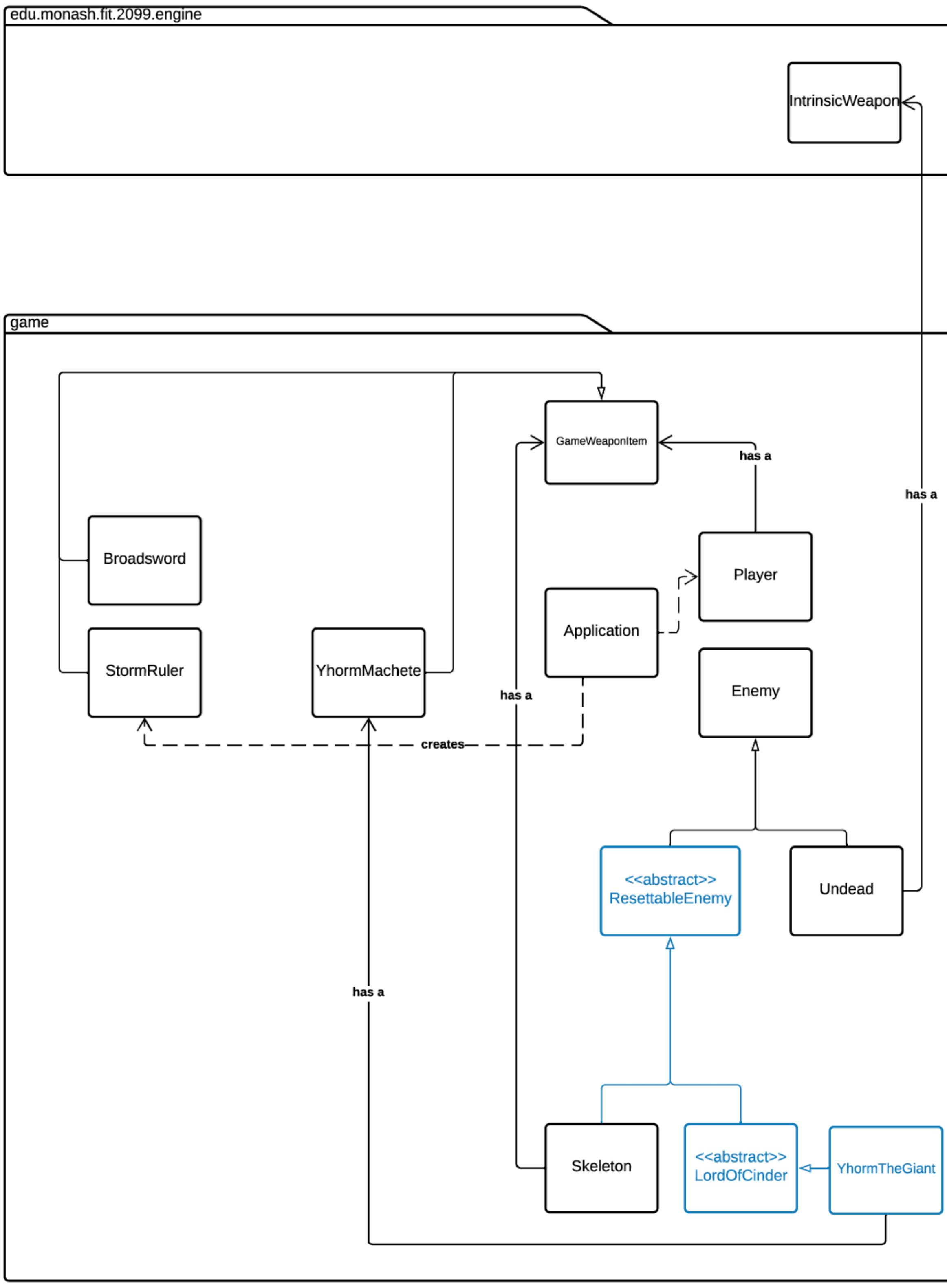
Weapons are created and added to the actor's inventory if he can carry a weapon. Each weapon will have its own class for instance, Broadsword class, GiantAxe class, StormRuler class, and YhormMachete class which extends GameWeaponItem class. These classes are created to enable customization for their own unique attribute, active skills, and passive skills. StormRuler is the weapon that can be equipped by the player that is placed next to Yhorm the Giant, thus the application creates an instance of StormRuler that is placed inside the game map. To enable switching the current weapon with the weapon on ground, PickItemAction can be overridden to execute SwapWeaponActionwhich automatically discards the current weapon and adds the other weapon to a player's inventory. In order to implement skills depending on type of skills, passive skills don't need creation of action as they can be directly done by overriding available methods accordingly. Active skills on the other hand should be created as a subclass of WeaponAction as different abilities require different implementation. The instance for subclasses of WeaponAction can be provided by adding abilities to specific weapons which will generate the instance of the WeaponAction. In addition to extending WeaponAction, these subclasses can provide constructor accepting combinations of weapon instance, target, and direction as parameters using Dependency Inversion and Injection to avoid downcasting. To add effects on actor, ground, or weapon itself for the active skills, capabilities can be added or removed accordingly and additional checks can be added in each game turn. These weapon actions can be set to be returned either from getPassiveSkills or getAllowableAction depending if the active skills require another actor or not.

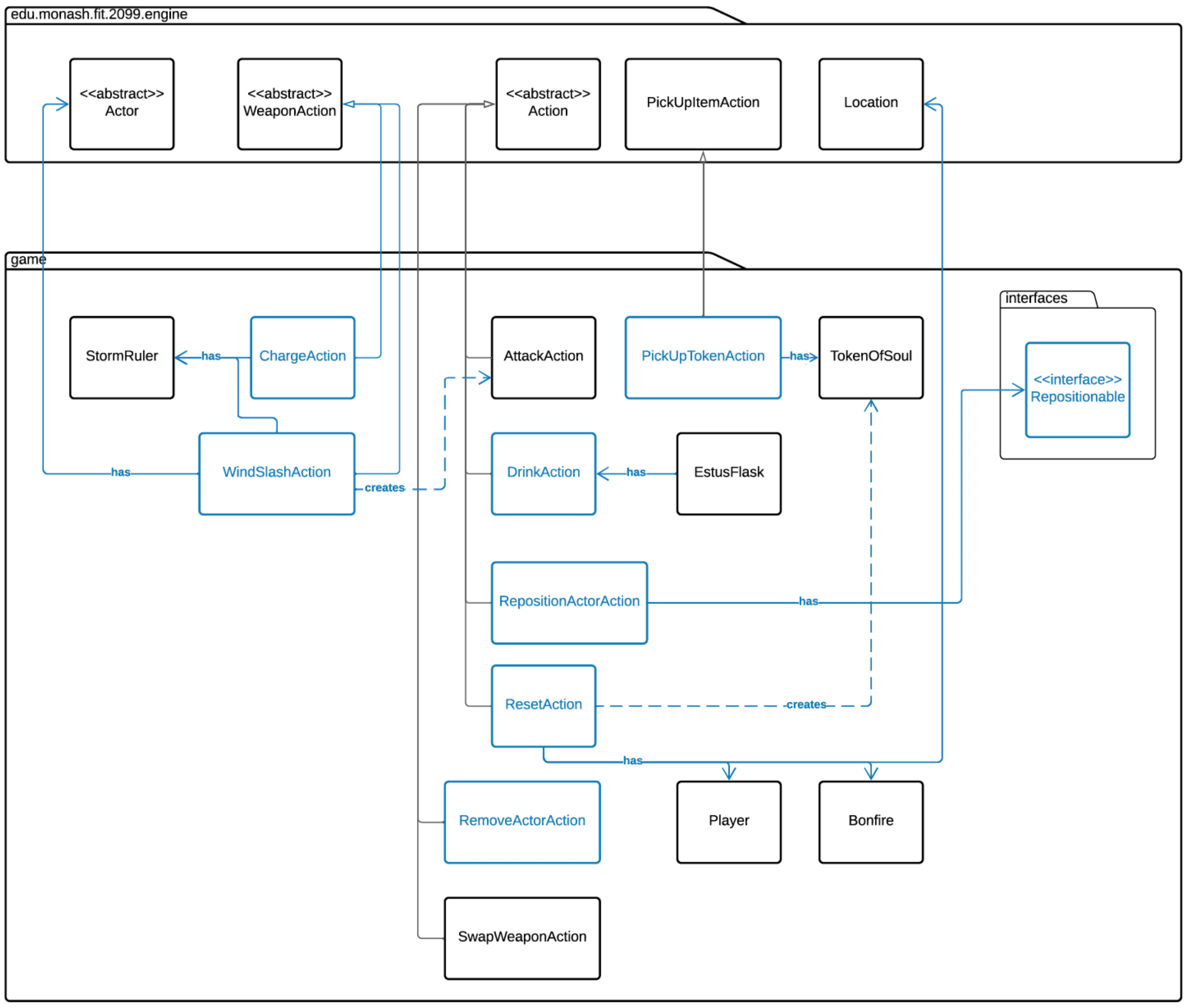
**Class Diagrams**



edu.monash.fit.2099.engine
<<abstract>> Actor
<<abstract>> Item
game
interfaces
TokenOfSoul
is a
is a
PortableItem
<<interface>> Drinkable
EstusFlask
<<interface>> Soul
has
1
has
1
Player
1
creates
1
<<interface>> Consumer
Application
Enemy
<<interface>> Resettable
is a
is a
<<interface>> Repositionable
Undead
<<abstract>> ResettableEnemy
has
CinderOfLord
is a
creates
8
Skeleton
<<abstract>> LordOfCinder
is a
YhormTheGiant
creates
1


edu.monash.fit.2099.engine
Location
<<abstract>> Ground
game
enums
Direction
Application
Player
Enemy
Undead
<<abstract>> ResettableEnemy
Bonfire
Cemetery
uses
has a
is a
is a
has reference to last bonfire rested
creates
creates
creates
1
1
1
1
1
5

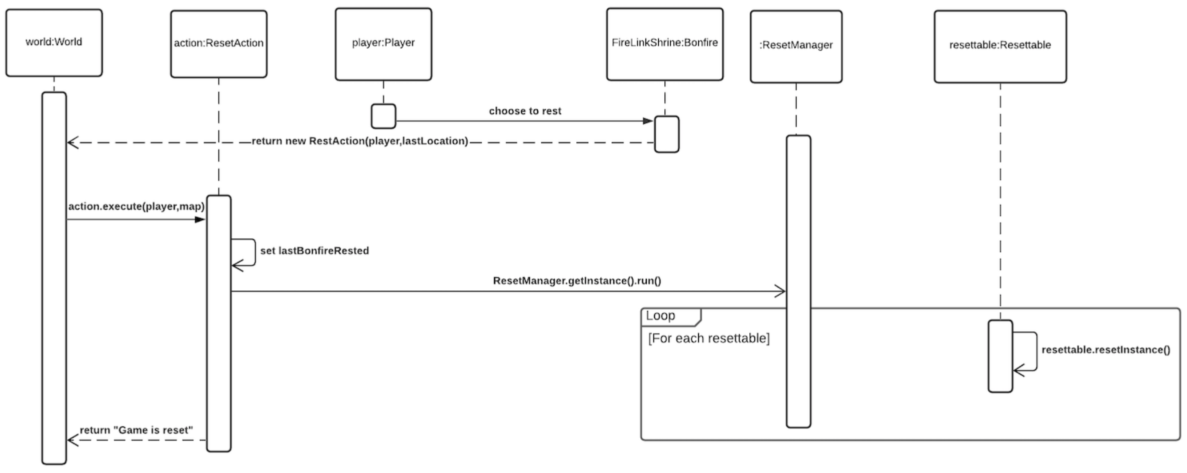

edu.monash.fit.2099.engine
IntrinsicWeapon
game
GameWeaponItem
has a
has a
Broadsword
Player
Application
StormRuler
YhormMachete
Enemy
has a
creates
<<abstract>> ResettableEnemy
Undead
has a
Skeleton
<<abstract>> LordOfCinder
YhormTheGiant



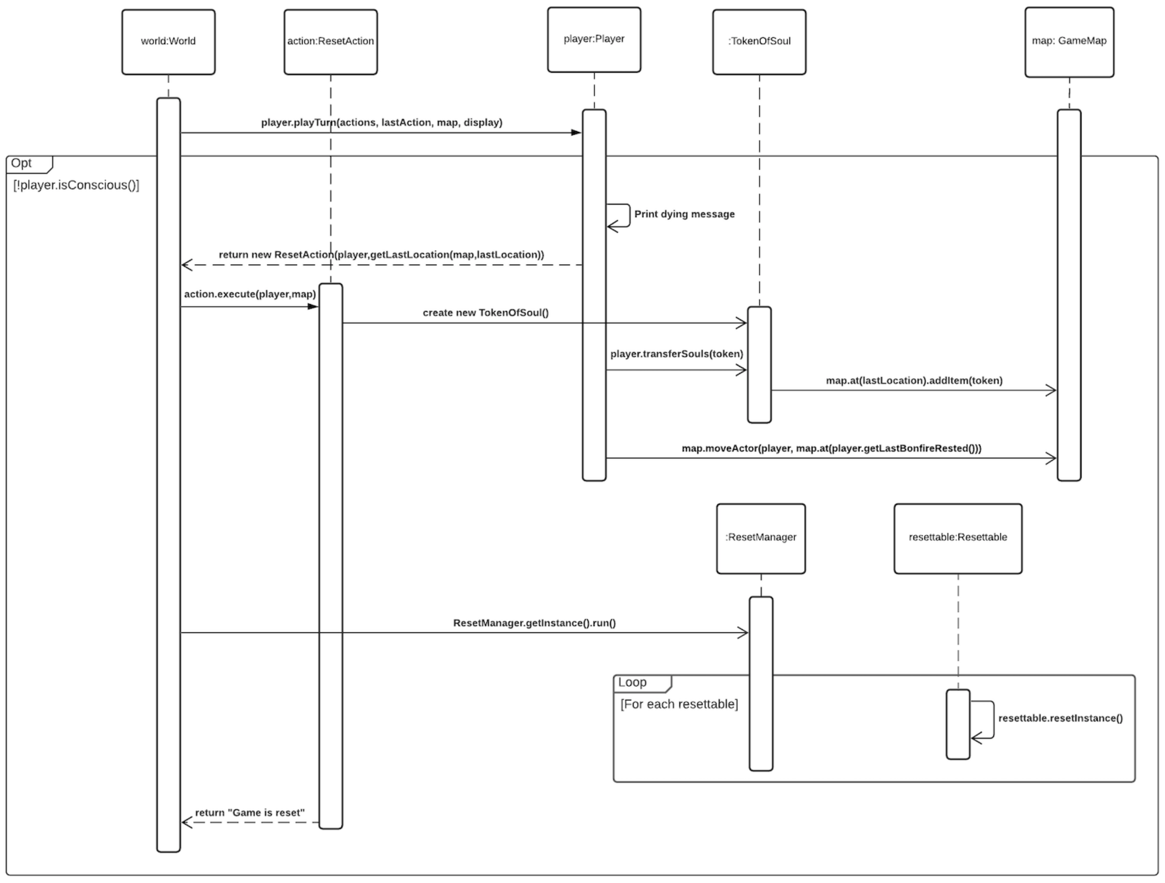
edu.monash.fit.2099.engine
<<abstract>> Actor
<<abstract>> WeaponAction
<<abstract>> Action
PickUpItemAction
Location
game
StormRuler
has
ChargeAction
AttackAction
PickUpTokenAction
has
TokenOfSoul
interfaces
<<interface>> Repositionable
has
WindSlashAction
creates
DrinkAction
has
EstusFlask
RepositionActorAction
has
ResetAction
creates
has
RemoveActorAction
Player
Bonfire
SwapWeaponAction

## Interaction Diagrams

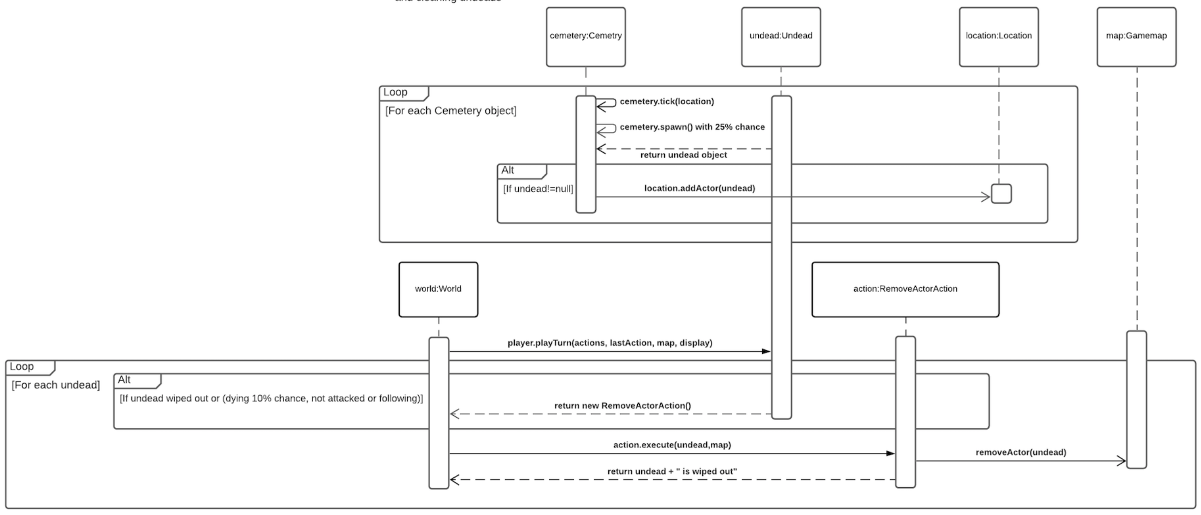
Player resting at bonfire sequence



Player die / soft reset sequence


world:World
action:ResetAction
player:Player
:TokenOfSoul
map: GameMap
player.playTurn(actions, lastAction, map, display)
Opt
[!player.isConscious()]
Print dying message
return new ResetAction(player,getLastLocation(map,lastLocation))
action.execute(player,map)
create new TokenOfSoul()
player.transferSouls(token)
map.at(lastLocation).addItem(token)
map.moveActor(player, map.at(player.getLastBonfireRested()))
:ResetManager
resettable:Resettable
ResetManager.getInstance().run()
Loop
[For each resettable]
resettable.resetInstance()
return "Game is reset"

Spawning undeads in cemetry
and cleaning undeads



cemetery:Cemetry
undead:Undead
location:Location
map:Gamemap
Loop
[For each Cemetery object]
cemetery.tick(location)
cemetery.spawn() with 25% chance
return undead object
Alt
[If undead!=null]
location.addActor(undead)
world:World
action:RemoveActorAction
player.playTurn(actions, lastAction, map, display)
Loop
[For each undead]
Alt
[If undead wiped out or (dying 10% chance, not attacked or following)]
return new RemoveActorAction()
action.execute(undead,map)
removeActor(undead)
return undead + " is wiped out"