MONASH University

FIT2099
Assessment 3
UML, Sequence Diagrams, Design Rationale

Team: LAB7_GROUP8

Team members:

1. **Tanul Gupta**
2. **Satoshi Kashima**
3. **Aditti Gupta**

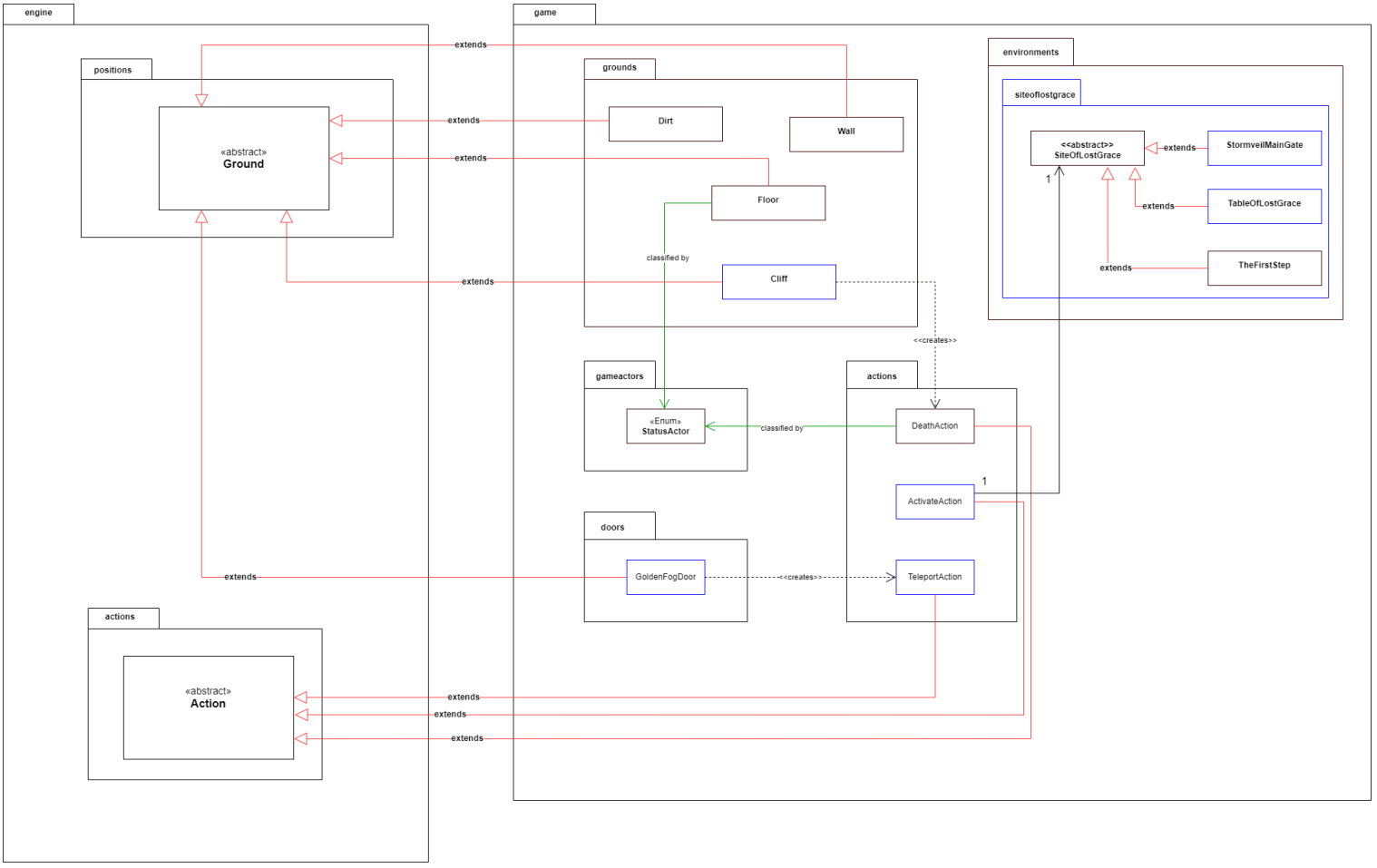
Date: 21/05/2023

Table Of Contents

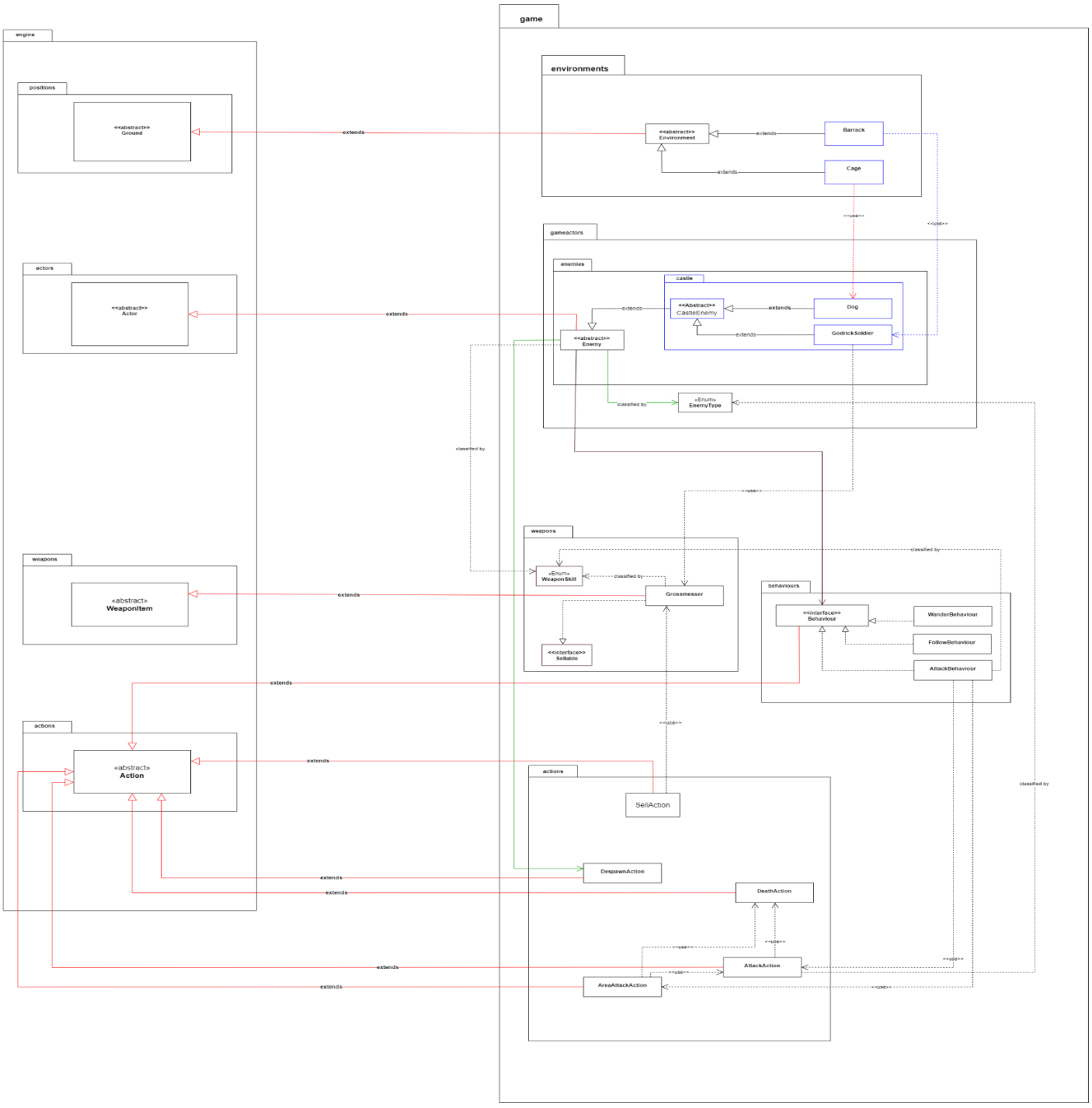
UML Diagrams:

Note: The classes that are different from A1 and A2 are highlighted in blue

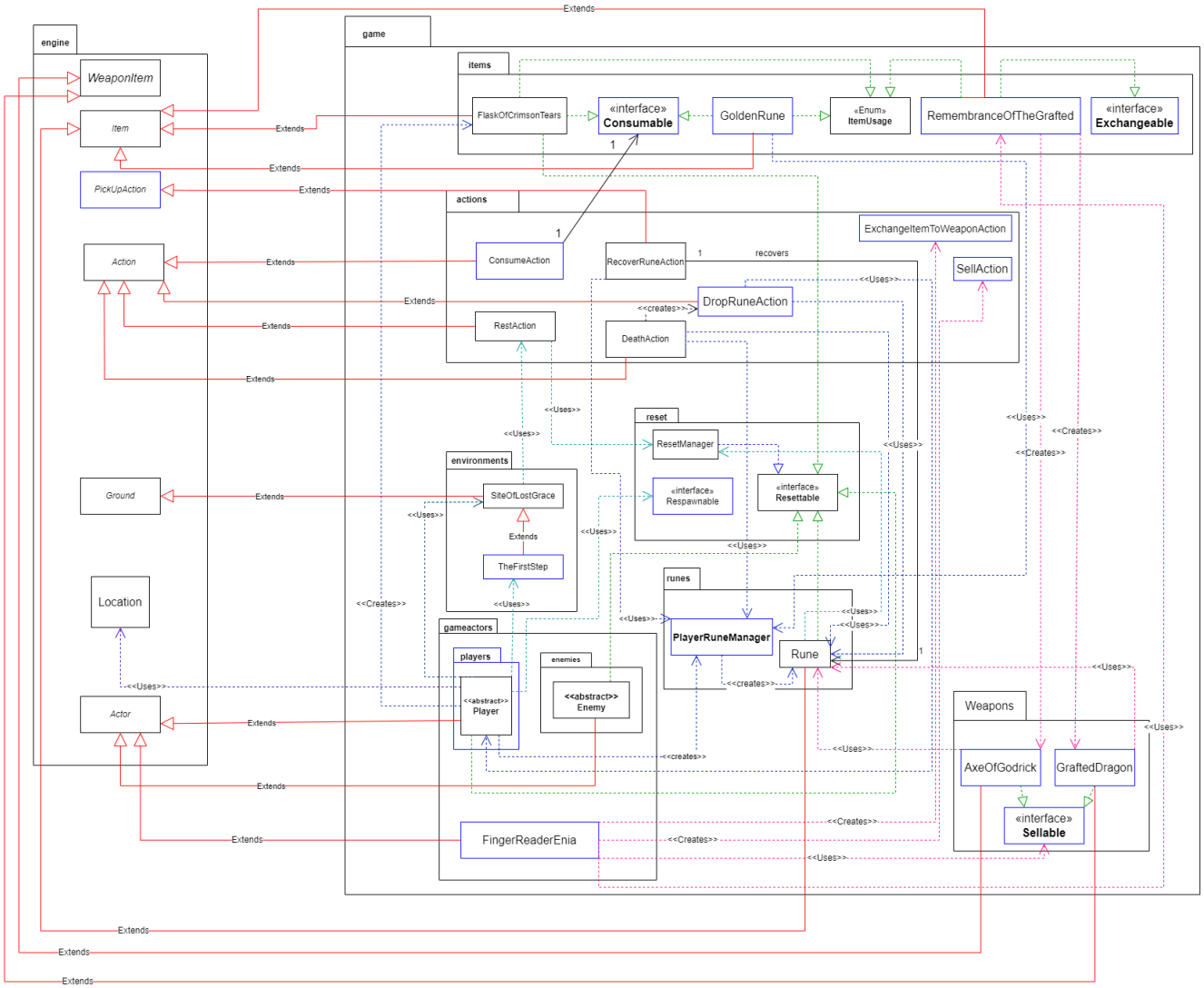
REQ1 UML:



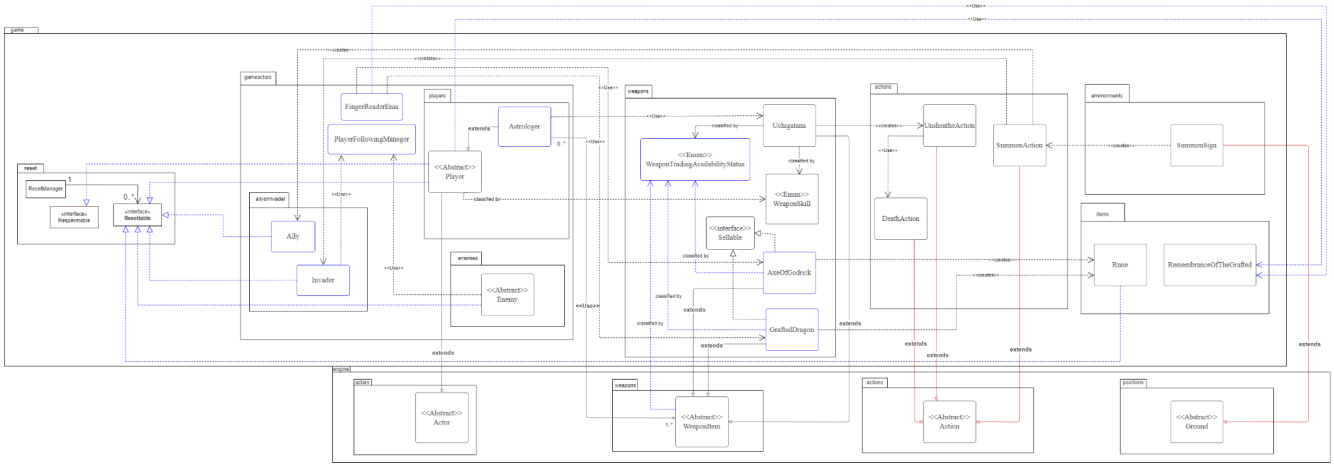
REQ2 UML:


engine
positions
«abstract»
Ground
actors
«abstract»
Actor
weapons
«abstract»
WeaponItem
actions
«abstract»
Action
game
environments
«abstract»
Environment
Barrack
Cage
gameactors
enemies
castle
«Abstract»
CastleEnemy
Dog
GodrickSoldier
«abstract»
Enemy
«Enum»
EnemyType
weapons
«Enum»
WeaponSkill
Grossmesser
«interface»
Sellable
behaviours
«interface»
Behaviour
WanderBehaviour
FollowBehaviour
AttackBehaviour
actions
SellAction
DespawnAction
DeathAction
AttackAction
AreaAttackAction
extends
classified by
«use»

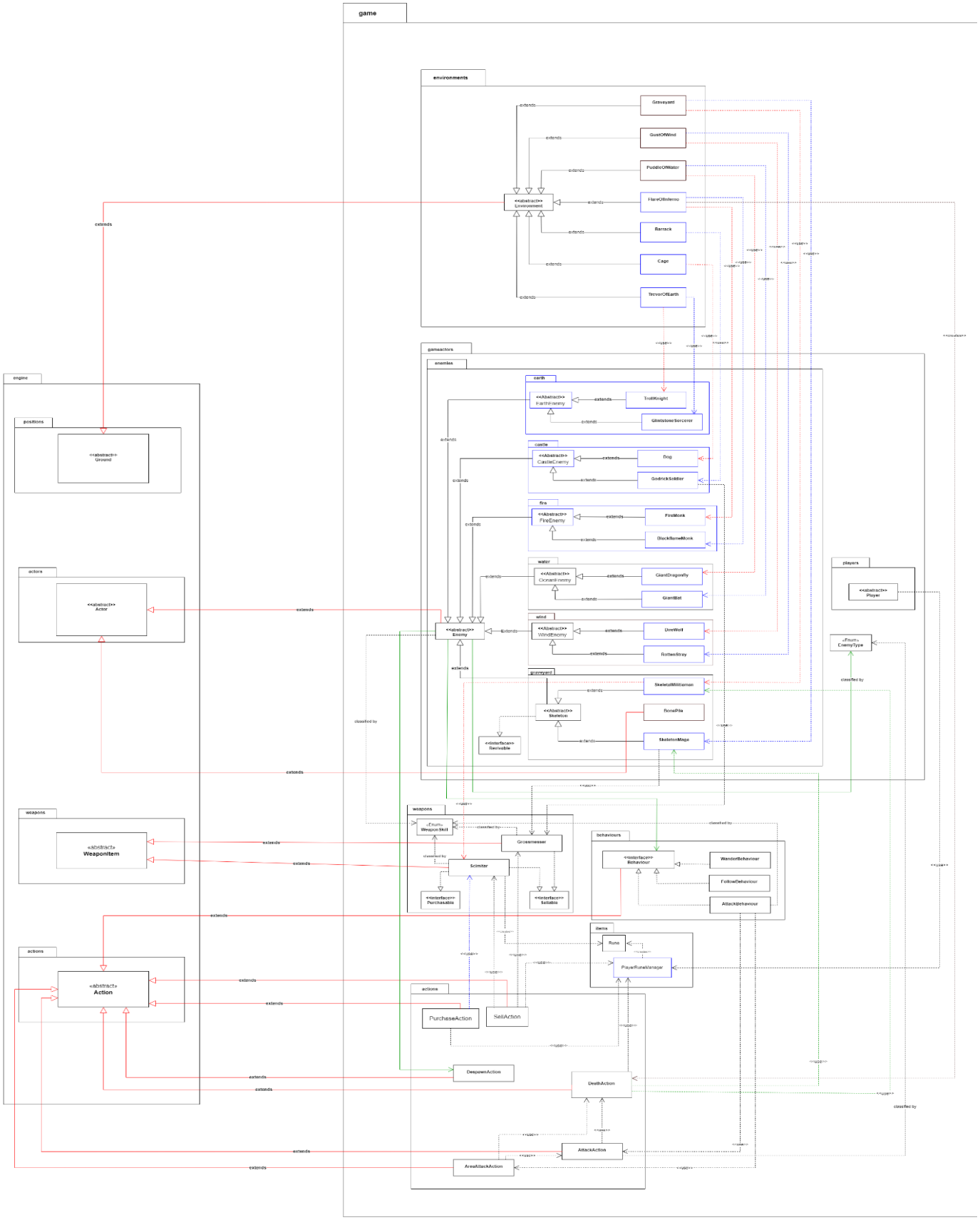
REQ3 UML:


engine
WeaponItem
Item
PickUpAction
Action
Ground
Location
Actor
game
items
FlaskOfCrimsonTears
«interface» Consumable
GoldenRune
«Enum» ItemUsage
RemembranceOfTheGrafted
«interface» Exchangeable
actions
ConsumeAction
RecoverRuneAction
recovers
ExchangeItemToWeaponAction
SellAction
DropRuneAction
RestAction
DeathAction
reset
ResetManager
«interface» Respawnable
«interface» Resettable
environments
SiteOfLostGrace
TheFirstStep
runes
PlayerRuneManager
Rune
gameactors
players
<<abstract>> Player
enemies
<<abstract>> Enemy
FingerReaderEnia
Weapons
AxeOfGodrick
GraftedDragon
«interface» Sellable
Extends
<<Uses>>
<<Creates>>
<<creates>>
1

REQ4 UML:


game
gameactors
players
FingerReaderEnia
PlayerFollowingManeger
Astrologer
<<Abstract>>
Player
reset
ResetManager
<<interface>>
Resettable
abyssinvader
Ally
Invader
enemies
<<Abstract>>
Enemy
weapons
Uchigatana
<<Enum>>
WeaponTradingAvailabilityStatus
<<Enum>>
WeaponSkill
<<interface>>
Sellable
AxeOfGodrick
GraftedDragon
actions
UnsheatheAction
SummonAction
DeathAction
environments
SummonSign
items
Rune
RemembranceOfTheGrafted
engine
actors
<<Abstract>>
Actor
weapons
<<Abstract>>
WeaponItem
actions
<<Abstract>>
Action
positions
<<Abstract>>
Ground

REQ5 UML:


game
environments
Graveyard
GustOfWind
PuddleOfWater
HaveOfInferno
Barrack
Cage
TrevorOfEarth
<<abstract>> Environment
gameactors
enemies
earth
<<Abstract>> EarthEnemy
TrollKnight
GlintstoneSorcerer
castle
<<Abstract>> CastleEnemy
Dog
GodrickSoldier
fire
<<Abstract>> FireEnemy
FireMonk
BlockRuneMonk
water
<<Abstract>> OceanEnemy
GiantDragonfly
GiantBat
wind
<<Abstract>> WindEnemy
DireWolf
RottenStray
graveyard
<<Abstract>> Skeleton
SkeletalMilitiaman
RonaPila
SkeletonMage
<<abstract>> Enemy
<<interface>> Revivable
players
<<abstract>> Player
«Enum» EnemyType
engine
positions
<<abstract>> Ground
actors
<<abstract>> Actor
weapons
«abstract» WeaponItem
actions
«abstract» Action
extends
classified by
weapons
«Enum» WeaponSkill
Grossmesser
Scimitar
<<interface>> Purchasable
<<interface>> Sellable
behaviours
<<interface>> Behaviour
WanderBehaviour
FollowBehaviour
AttackBehaviour
items
Runs
PlayerRuneManager
actions
PurcheseAction
SellAction
DespawnAction
DeathAction
AttackAction
AreaAttackAction
<<use>>

Assignment 3 Requirements Design Rationale

REQ 1

Cliff:

In summary, the Cliff class provides a specialized representation of a cliff ground type in the game. It extends the Ground class, overrides the tick() method to trigger a death action for the player, and encapsulates cliff-specific behavior. The design promotes code reuse, encapsulation, and modularity.

Golden Fog Door:

The GoldenFogDoor class represents a special door in the game that allows only the player character to enter. It extends the Ground class and imports necessary packages and classes.

The class has a constructor that sets the visual representation of the door.

The canActorEnter() method checks if the actor is a player, allowing only players to enter the door.

The allowableActions() method adds a TeleportAction to the list of actions if the actor is a player.

Overall, the GoldenFogDoor class restricts entry to the player character and provides a TeleportAction as the allowable action for the player.

Teleport Action:

The TeleportAction class represents an action that allows an actor, specifically the player, to teleport to different locations in the game. It extends the Action class and includes a static HashMap to store the locations of fog doors.

The addFogDoorLocation() method adds fog door locations to the HashMap. It takes a location name and the corresponding location of the endpoint as parameters and stores them in the fogDoorLocations HashMap. `"LIMGRAVE_ROUNDTABLE"` means door in Limgrave to Door in Roundtable.

The execute() method is overridden to perform the teleportation action. It checks the current location of the actor and moves them to the appropriate endpoint location based on specific conditions and the fogDoorLocations HashMap.

The menuDescription() method describes the action displayed in the game menu.

The Application class serves as the main class to start the game. It creates the game world, initializes game maps, sets up the player character, and adds game elements such as ground types, fog doors, and enemy behaviours.

Site Of Lost Grace (Optional)

The SiteOfLostGrace class is designed as an abstract class, indicating that it serves as a base class for specific types of lost grace sites.

- The class includes instance variables to track the site's activation and discovery status.
- The siteLocation variable represents the location of the site.
- The tick() method updates the siteLocation based on the given location.
- The allowableActions() method returns a list of actions that can be performed at the site, depending on the actor's capabilities and the activation and discovery status of the site.
- The canActorEnter() method determines whether an actor (specifically, a player) can enter the site.
- The ActivateAction class represents the action of activating a SiteOfLostGrace.
- The class includes a constructor that takes a SiteOfLostGrace object as a parameter.
- The execute() method sets the activation status of the site to true.

The player can only Rest if SiteOfLostGrace is activated and, by default, TheFirstStep is activated. The other two sites StormveilMainGate and the TableOfLostGrace, will display Lost Grace Discovered when the player first comes in its exits then the player can activate the grace, which will then allow them to rest at the grace, which will set their respawn point to that grace.

Pros of this approach:

- Modularity: The use of abstract classes and separate action classes allows for easy extension and customization of the game environment and actions.
- Encapsulation: The class encapsulates the behavior and properties specific to a SiteOfLostGrace, making it easier to manage and modify the behavior of this type of environment.
- Flexibility: The design allows for different types of lost grace sites to be created by extending the base class and overriding methods as needed.

REQ 2

Environments:

Environment (abstract class):

In the package named environments, an abstract class called Environment that extends Ground from engine is made. It contains a constructor and an abstract method for spawning an Enemy on a GameMap. It also has a method to determine if a given Location is in the eastern half of the GameMap, to differentiate spawning for different enemies with their respective locations, i.e, East or West (specified below). The Environment class is designed to be a base class for various environments in the game (Graveyard, GustOfWind and PuddleOfWater), where enemies can spawn.

Pros:

SOLID Principles Used:

- Single Responsibility Principle (SRP): The Environment class has a single responsibility of providing a base class for environments in the game, with the capability to spawn enemies. The tick method is responsible for checking for the presence of enemies and spawning them if none are found. The spawn method is responsible for actually creating an enemy in the given Location and GameMap.
- Open-Closed Principle (OCP): The Environment class is designed to be open for extension and closed for modification. The class is designed to be extended by subclasses to implement the actual spawning logic. The Environment class itself does not need to be modified for this.
- Liskov Substitution Principle (LSP): Since Environment is an abstract class, it cannot be instantiated, but it can be used as a type for its subclasses. This means that wherever an Environment object is expected, any of its subclasses can be used without affecting the correctness of the program.
- Dependency Inversion Principle (DIP): The Environment class depends on the GameMap, Location, Ground, Display, and RandomNumberGenerator classes/interfaces to function. However, it does not create any of these dependencies directly. Instead, they are passed as parameters to the spawn and tick methods. This allows for greater flexibility in choosing the implementation of these dependencies at runtime.

Cons and improvements:

- S - Single Responsibility Principle: The Environment class should have only one responsibility. The current implementation has the responsibility of spawning an Enemy as well as determining if a location is in the eastern half of the GameMap. To adhere to the SRP, we should split these responsibilities into two separate classes.
- O - Open-Closed Principle: The Environment class should be open for extension but closed for modification. Currently, the spawn method is abstract, which allows for subclasses to extend the functionality, but the tick method is not open for extension. We can modify the tick method to be open for extension by refactoring it to call a separate method that subclasses can override.

The Cage (spawns Dog) and Barrack (spawns GodrickSoldier) classes are subclasses of the Environment abstract class in a game. They represent specific types of game environments where enemies can spawn.

Enemies:

Enemy (abstract class):

This is an abstract class Enemy that extends Actor, which represents an enemy in a game. It has a constructor that takes the name, display character, and hit points of the enemy, and sets its capabilities as an enemy. It also contains a list of behaviours with their priorities, an intrinsic weapon, and methods to determine the allowable actions for the enemy and if it can target other actors.

Pros:

SOLID Principles Used:

- Single Responsibility Principle: The Enemy class has a single responsibility of representing an enemy in a game and its associated behaviors.
- Open-Closed Principle: The Enemy class is designed to be extended for different types of enemies and their behaviors, making it open for extension but closed for modification.
- Liskov Substitution Principle: The Enemy class is a subtype of the Actor class and can be used wherever an Actor is expected.
- Interface Segregation Principle: The Enemy class only exposes methods that are necessary for its behaviors and interactions with other game objects.
- Dependency Inversion Principle: The Enemy class depends on abstractions such as the Behaviour interface and the Resettable interface, making it easier to modify and extend.

Cons and improvements:

- The allowableActions() method should take the map and player as arguments, rather than assuming that the player is a global variable accessible by the Enemy class.
- The intrinsicWeapon and behaviours Map should be declared as private and have public getter and setter methods to access and modify them.

CastleEnemy:
The CastleEnemy class is designed as an abstract class that extends the Enemy class and implements the Resettable interface. It represents an enemy specific to the castle in a game. The constructor initializes the properties inherited from the Enemy class and sets the capabilities of the enemy. It also registers the enemy as a Resettable object.

Dog:
Dog extends the CastleEnemy class. The Dog class has a constructor that initializes the name, symbol, health, attack, and defense values of the enemy. It also overrides the getIntrinsicWeapon() method from the parent class to return a specific intrinsic weapon for the Dog.

GodrickSoldier:
GodrickSoldier class extends the CastleEnemy class. It is used to create instances of Godrick's soldiers in a castle-themed game. The constructor initializes the attributes of the GodrickSoldier object, including the name, type, health, attack power, and defense power. It also adds a weapon called Grossmesser to the soldier's inventory.

Changes From Assignment 2 Feedback:

- Added PlayerRuneManager which handles all the modifications that happen throughout the game to the Players Rune.
- PlayerRuneManager keeps track of the Runes dropped by player on death.
- Rune class now only defines what a Rune is and it's properties.

PlayerRuneManager:

The PlayerRuneManager class is designed to manage the runes possessed by the player in a game. It provides methods for increasing and decreasing the amount of a specific type of rune, as well as resetting the player's rune to its initial state. Let's discuss the pros and cons of using this class:

Pros:

Encapsulation: The PlayerRuneManager encapsulates the logic and operations related to managing player runes. It keeps the rune-related functionality separate from other game components, promoting modular and organized code.

Singleton Design Pattern: The PlayerRuneManager uses the Singleton design pattern, ensuring that there is only one instance of the class throughout the game. This design pattern provides global access to the manager, making it convenient to handle player runes from anywhere in the codebase.

Centralized Management: By having a dedicated class for managing player runes, it becomes easier to track and modify the player's rune state. Other parts of the game can interact with the PlayerRuneManager to perform rune-related actions, such as increasing or decreasing the rune count.

Reset Functionality: The PlayerRuneManager includes a resetPlayerRune() method that resets the player's rune to its initial state. This can be useful in scenarios where the game needs to restart or when a player's progress is reset.

Cons:

Limited Flexibility: The PlayerRuneManager class is tailored specifically for managing player runes. It may not be suitable for handling more complex rune-related functionality or interacting with other game systems. Additional design considerations and modifications may be required to adapt the class to evolving game requirements.

SOLID Principles:

Single Responsibility Principle (SRP): The PlayerRuneManager class adheres to SRP by having a single responsibility of managing player runes. It handles operations such as increasing, decreasing, and resetting the rune count, without taking on additional responsibilities.

Dependency Inversion Principle (DIP): The PlayerRuneManager class does not have explicit dependencies on other classes. It relies on the Rune class, which is a general item representation. This loose coupling allows for flexibility and easy substitution of the Rune object with other item types if needed.

In summary, the PlayerRuneManager class provides a centralized and encapsulated approach for managing player runes in a game. It adheres to the SOLID principles of SRP and DIP, promoting maintainability and modularity. However, it should be enhanced with proper error handling and be flexible enough to accommodate future changes in the game's rune system.

REQ 3

Design/Implementation changes from Assignment 2:

1. Added Consumable Interface, which consumables will be implemented by items in game that are consumable such as FlaskOfCrimsonTears and GoldenRune
2. Respawn will now add the starting weapon for example Uchigatana back to the player's inventory if they sold it or dropped it somewhere before dying.

Consumables:

Consumable Interface:
The Consumable interface defines the contract for items consumed in the game.
It includes two methods:

1. String consume(Actor actor): Represents the action of consuming the item by an actor. It returns a string indicating the outcome of the consumption.
2. Boolean consumeBy(Actor actor): Indicates whether a specific actor can consume the item. It returns a boolean value.

Using this interface, different items in the game can implement their consumption behaviour while adhering to a common interface.

ConsumeAction Class:

- The ConsumeAction class extends the Action class provided by the game engine.
  It represents the action of consuming a consumable item.
- The constructor takes a Consumable object as a parameter, which allows the action to be associated with a specific consumable item.
- The execute() method checks if the actor can consume the item (consumeBy(actor)), and if so, it invokes the consume(actor) method of the consumable item. Otherwise, it returns a string indicating that the actor does nothing.

The implementation of Consumables and ConsumeAction demonstrates adherence to some of the SOLID principles:

Single Responsibility Principle (SRP):

The Consumable interface has a single responsibility: defining the contract for consumable items.
The ConsumeAction class also follows SRP by having a single responsibility: executing the action of consuming a consumable item.

Open-Closed Principle (OCP):
The Consumable interface and ConsumeAction class are open for extension but closed for modification.

Adding new types of consumable items can be added by implementing the Consumable interface without modifying the existing code.
Similarly, new types of actions related to consuming can be added without modifying the ConsumeAction class.

Interface Segregation Principle (ISP):

The Consumable interface has segregated methods related to consumption behaviour.
This allows implementers only to implement the necessary methods based on the requirements of their specific consumable items.

Flask Of Crimson Tears:

The class represents an item that can be consumed to heal an actor's hit points in the game.
It implements the Resettable and Consumable interfaces.
The class includes the following key features:

- **Initialization:** The constructor sets the item's name, display character, and other properties. It registers the item as resettable and adds the ConsumeAction to the available actions.
- **Healing Amount and Maximum Consume Amount:** The class defines the amount of hit points the item heals (HEAL_AMOUNT) and the maximum number of times it can be consumed (MAX_CONSUME_AMOUNT).
- **Consumed Count**: The class keeps track of the number of times the item has been consumed (consumedCount).
- **Resetting:** The class implements the Resettable interface to reset the consumed count.
- **Availability:** The class provides methods to check if the item is available for consumption.
- **Consumption:** The class implements the Consumable interface methods to handle consumption by an actor, including healing the actor and updating the consumed count.

Golden Rune:

The GoldenRune class represents a consumable item in the game. It extends the Item class and imports necessary packages and classes. This item can only be consumed when in the Players inventory, and so the consume action is initialised in the Player class playTurn method.

The class has the following instance variables:

- consumed: a boolean indicating whether the GoldenRune has been consumed.
- MAX_AMOUNT and MIN_AMOUNT: constants representing the maximum and minimum amounts of runes obtainable.

The constructor initializes the GoldenRune with a name, character, and portability. It adds the ItemUsage.IS_GOLDEN_RUNE capability.

- The isAvailable() method checks if the GoldenRune is available for consumption.
- The updateStatus() method marks the GoldenRune as consumed.
- The getRuneAmount() method generates a random amount of runes.

- The consume() method allows the player to consume the GoldenRune, increasing their rune inventory and updating the status. It also removes the GoldenRune from the player's inventory.
- The consumeBy() method checks if the actor is a player and if the GoldenRune is available for consumption.

Overall, the GoldenRune class provides functionality for consuming the item and handling related actions.

Finger Reader Enia:

Main Features:
Represents a special type of trader in the game. Inherits from the Actor class.
Capabilities:
- IS_TRADER: Indicates that this actor can engage in trading activities.
- CANNOT_BE_ATTACKED: Signifies that this actor cannot be attacked.

allowableActions() method:
- Determines the actions that the FingerReaderEnia actor can perform towards another actor. Allows exchange of a specific item (RemembranceOfTheGrafted) with the player using ExchangeItemToWeaponAction.
- Enables the FingerReaderEnia to purchase weapons from the player using SellAction.

The FingerReaderEnia class represents a special trader actor that can exchange a specific item (RemembranceOfTheGrafted) with the player and sell weapons to the player. It cannot be attacked by other actors.

Axe of Godrick:

The AxeOfGodrick class represents an axe item called "Axe of Godrick" that extends the WeaponItem class and implements the Sellable interface.

- **Extension:** The class extends the WeaponItem class, allowing it to inherit properties and behavior from the parent class.
- **Interface Implementation:** The class implements the Sellable interface, indicating that it can be sold in the game.
- **Constructor:** The constructor initializes the AxeOfGodrick object. It sets the name of the axe to "Axe of Godrick", assigns the symbol 'T', and sets the damage points to 142. The axe has a damage type of "chop" with a damage value of 84. Additionally, it is capable of performing targeted attacks.

The rationale behind this class is to represent a specific axe item called "Axe of Godrick" in the game. The axe has its own unique characteristics, such as damage points, damage type, and the ability to perform targeted attacks. It is also sellable to a specific actor (FingerReaderEnia). The AxeOfGodrick class demonstrates the extensibility and customization of weapons in the game.

Grafted Dragon:

The GraftedDragon class represents a weapon item called "Grafted Dragon" that extends the WeaponItem class and implements the Sellable interface.

- **Extension:** The class extends the WeaponItem class, inheriting properties and behavior from the parent class.
- **Interface Implementation:** The class implements the Sellable interface, indicating that it can be sold in the game.
- **Constructor:** The constructor initializes the GraftedDragon object. It sets the name of the weapon to "Grafted Dragon", assigns the symbol 'N', and sets the damage points to 89. The weapon has a damage type of "bite" with a damage value of 90. It is also capable of performing targeted attacks.

The rationale behind this class is to represent a specific weapon item called "Grafted Dragon" in the game. The weapon has its own unique characteristics, such as damage points, damage type, and the ability to perform targeted attacks. It is also sellable to a specific actor (FingerReaderEnia). The GraftedDragon class demonstrates the extensibility and customization of weapons in the game.

Exchangeable Interface:

The Exchangeable interface represents an entity that can provide a list of available weapons for exchange.

Interface: The Exchangeable interface defines the contract for classes that can provide a list of available weapons for exchange.
getAvailableWeaponsForExchange() method: The interface includes a method getAvailableWeaponsForExchange() that retrieves the list of available weapons for exchange. It returns a list of WeaponItem objects.

The rationale behind this interface is to provide a common structure for entities that can offer a selection of weapons for exchange. The Exchangeable interface ensures that classes implementing it will provide the necessary functionality to retrieve the list of available weapons for exchange.

Remembrance of the Grafted:

The RemembranceOfTheGrafted class represents a special item called "Remembrance of the Grafted" in the game. It is a subclass of the Item class and implements the Exchangeable interface. The class also contains a list of available weapons for exchange.

Inheritance: The class extends the Item class, allowing it to inherit properties and behavior from the parent class.
Interface Implementation: The class implements the **Exchangeable interface**, indicating that it can be exchanged in the game.
Constructor: The constructor initializes the RemembranceOfTheGrafted object. It sets the name of the item to "Remembrance of the Grafted", assigns the symbol 'O', and sets it as non-disposable.

Additionally, it adds the capability of being exchangeable. The constructor also initializes the availableWeaponsForExchange list and adds specific weapon objects (AxeOfGodrick and GraftedDragon) to the list.
getAvailableWeaponsForExchange() method: The getAvailableWeaponsForExchange() method is overridden from the Exchangeable interface. It returns the list of available weapons for exchange. The rationale behind this class is to represent a special item in the game that can be exchanged for specific weapons (AxeOfGodrick and GraftedDragon). The RemembranceOfTheGrafted class provides the functionality to retrieve the list of available weapons for exchange.

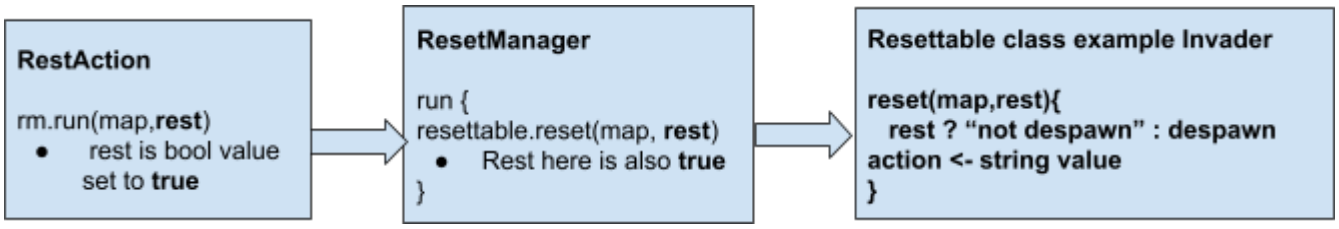
Reset on Rest:

NON-OPTIONAL CHANGE:

- Method signature of reset changed to reset(GameMap map, boolean rest): The reset method now takes two parameters. The first parameter is the GameMap object, and the second parameter is a boolean value rest that indicates whether the reset is being triggered due to a rest action.
- Added boolean value to indicate reset triggered by rest: By including the rest parameter, it explicitly specifies whether the reset is happening because of a rest action. This provides clarity and allows different handling for resets triggered by rest and other conditions (such as player death).
- Removal of resettable from the list: In the run method, the resettable is now removed from the list of resettable only if it can be removed on rest. This ensures that the resettable is only removed in the appropriate scenario.

Overall, these changes enhance the clarity and flexibility of the reset method by incorporating a boolean value to differentiate between resets triggered by rest and other conditions. This allows for specific handling of resets in different scenarios.

FOR SIMPLICITY:



Pros:

Flexibility: Adding the boolean flag distinguishes between resets triggered by rest actions and other actions, such as player death. This enables different handling of resettable objects based on the specific scenario.

Fine-grained Control: Passing the rest flag to the reset() method of resettable objects can customize the reset behaviour based on the rest condition. This allows for more precise control over the reset process.
Modular Design: The approach maintains a modular design by keeping the responsibility of resetting objects within the ResetManager class while allowing individual resettable objects to have specific behaviours based on the rest condition.

Cons:

- **Increased Complexity:** Introducing the boolean flag and handling different scenarios can add complexity to the code. It requires careful management of the flag and its logic to ensure the proper functioning of the reset process.
- **Potential Confusion:** How the rest flag is utilized may introduce confusion or make the code harder to understand. Developers must ensure the flag's usage is clear and consistent throughout the codebase.
- **Possible Maintenance Challenges:** If the reset logic becomes more complex or additional conditions are introduced in the future, the code may become harder to maintain and modify.

Overall, the pros of this approach outweigh the cons, as it provides more flexibility and control over the reset process. Therefore we have clear documentation of how this works.

Changes from Assignment 2 (Made for ASSIGNMENT 3 REQ1 SITE OF LOST GRACE OPTIONAL):

- Overall implementation is the same, with some changes to RestAction where we set the player to respawn at the Site Of Lost Grace the player is resting at.

The reset feature has been implemented to make it extensible for future resettables. To achieve this, we have created a Resettable interface which defines the behaviour for resetting a component or system. Any class that implements this interface can be reset using the reset feature.

In addition to the Resettable interface, we have created a ResetManager class which maintains a list of all the resettable components and provides a single access point for resetting them. This design allows new resettables to be added to the system without modifying the reset feature code itself, thus making it more extensible.

To further enhance the extensibility of the reset feature, we have made it possible for resettables to register themselves with the ResetManager class. This means that new resettables can be added to the system dynamically without requiring any modifications to existing code.

Overall, the reset feature has been designed with extensibility, allowing it to be easily adapted and expanded as new resettables are added to the system. Using the Resettable interface and the ResetManager class, we have created a flexible and robust reset feature that can accommodate future changes and additions to the system.

The Resettable interface has the following advantages regarding SOLID principles:

- Single Responsibility Principle (SRP): The Resettable interface defines the behaviour for resetting a component or system. This helps to keep the interface simple and focused on a single task.

- Open-Closed Principle (OCP): The ResetManager class is designed to be open for extension but closed for modification. It achieves this by maintaining a list of Resettable objects and providing a single access point for resetting them. This design allows new resettable's to be added to the system without modifying the ResetManager code.

- Liskov Substitution Principle (LSP): Any class that implements the Resettable interface can be used interchangeably with other Resettable objects. This means that the behaviour of the ResetManager class is consistent regardless of the type of Resettable object being reset.

- Interface Segregation Principle (ISP): The Resettable interface is designed with only the necessary methods for resetting a component or system. This helps prevent clients from being forced to implement methods they do not need.

- Dependency Inversion Principle (DIP): The ResetManager class depends on the Resettable interface rather than concrete implementations of Resettable objects. This allows the ResetManager class to work with any object that implements the Resettable interface, promoting flexibility and extensibility.

Pros:

- ☐ Simplicity: The Resettable interface focuses on a single task, making it easier to understand and maintain.
- ☐ Consistency: The Liskov Substitution Principle ensures that any class that implements the Resettable interface can be used interchangeably, providing consistent behaviour for the ResetManager class.
- ☐ Separation of Concerns: The Interface Segregation Principle helps prevent clients from being forced to implement unnecessary methods, promoting better separation of concerns.
- ☐ Dependency Inversion: The Dependency Inversion Principle allows the ResetManager class to work with any object that implements the Resettable interface, promoting flexibility and modularity

Cons:

- ☐ Complexity: The design of the reset feature could be seen as more complex than a simpler approach, making it harder to understand for developers unfamiliar with it.

.

Respawning:

The respawn method in the Player class is designed to reset the player's hit points to their initial value, which is the player's maximum hit points when it is first created. This is accomplished by calling the resetMaxHp method inherited from the Actor class.

The reset method is called from the respawn method whenever the player dies or rests at the Site of Lost Grace. This method is responsible for resetting the player's health and returning a string message to indicate that the player's health has been reset.

The Player class implements the Respawnable interface and defines the respawn method. This interface is used to mark actors that can be respawned in the game.

The Resettable interface is also implemented by the Player class, which is used to register the player with the ResetManager singleton class. The ResetManager class is responsible for registering all resettable objects in the game and resetting them to their initial state when the game is reset.

Pros:

- ☐ The code follows the SOLID principles, making it easier to understand, maintain, and extend over time.
- ☐ The Respawn class is simple and easy to use, with a clear purpose and limited responsibility.

Cons:

- ☐ The Respawn class does not handle error checking or exception handling, which could lead to unexpected behaviour if errors occur.

REQ 4

Requirements:

- Implementing Astroger
- Implementing Summon Environment
- Implementing Ally and Invader

Implementation Details - Summon Environment

- SummonSign class extends the Environment class
- If a player steps on this ground, SummonAction is triggered
- SummonAction inherits the Action class. In its execution method, it randomly chooses who to Spawn, Ally, or Invader, using getRandomInt function from RandNumberGenerator class.
- When instantiating Ally or Invader, we use the static method of Player class to obtain the player's max health, which becomes the max health of Ally/Invader.
- This means that we have a new static variable that stores the Max Health of the player. Although this implies that we have two duplicates of Max Health value in Player's class, this cannot be helped since we cannot modify the Actor class.
- We achieve SRP by splitting the detection of a player in the Summon Environment and the actual procedure of Summoning.

Implementation Details - NPC, Ally and Invader

- To handle all NPCs, including Enemy, Ally, Invader, and any other NPCs that we might want in the future, we created an abstract NPC class. This class has a list of behaviors as an attribute and implements playTurn method which traverses through the list. This class has a static variable to store the player since NPC requires the player oftentimes (e.g. when instantiating certain behaviors).
- To classify friendly enemies to the player as a whole, we use an enumeration value of HOSTILE_TO_ENEMY. Both the Player and the Ally have this enumeration status. Oppositely, to classify any actors that can attack players as HOSTILE_TO_PLAYER. Both Enemy and Invader have this status. In canTarget method of each NPC, these two enumeration values allow an easier check to whether an NPC can target a subjected NPC.

Implementation Details - PlayerFollowingManager

- To handle the following status of NPC that can follow the Player, we created PlayerFollowingManger class. This class has a static variable to store the player and has a method that decides if a particular enemy should start following or not.
- By using this class, even if there is a new NPC class that requires the following behavior, we can just put an instance of the PlayerFollowingManager class as an attribute and invoke the updateFollowingStatusIfNeeded method inside playTurn method. Hence, we are adhering to DRY and SRP. We are also separating the concerns. We are also following the OCP since if there is a change in when/how to start following the player in the future, we can just create another manager, without modifying each NPC class.

REQ5

Summary of creative requirement:

- Divided the map into 4 parts - north-east, north-west, south-east and south-west
- Added two new types of enemies - fire (FireMonk, BlackflameMonk) and earth (TrollKinght, GlintstoneSorcerer)
- Added 2 new enemies in each type - water (GiantDragonfly, GiantBat), wind (DireWolf, RottenStray) and graveyard (SkeletalMilitiaman, SkeletonMage)
- Added 2 new spawning grounds - FlareOfInferno and TrevorOfEarth
- Added implementation of player dying if he steps on the spawning ground of fire enemy and will not be able to retrieve runes either (as he and the runes burn off)
- Added implementation of getting max hp if the player kills the earth enemy

The FlareOfInferno and TrevorOfEarth classes are subclasses of the Environment abstract class in a game. They represent specific types of game environments where enemies can spawn. Each class overrides the spawn method to randomly generate and return an instance of one or two types of enemy objects.

FlareOfInferno:
The FlareOfInferno class extends the Environment class, indicating that it represents a specific type of environment in a game. The class has two main responsibilities: spawning enemies and handling the tick operation when a player steps on the spawning ground.
The tick method is responsible for handling the ticking operation when a player steps on the FlareOfInferno spawning ground. It first calls the superclass's tick method, indicating that any default environment behavior should be executed. Then, it checks if the actor on the given location is the player. If it is, it creates a new DeathAction object, executes the action with the actor and the location's game map, and prints the result using the Display class.

Enemy spawning areas:
North: FireMonk
South: BlackflameMonk

TrevorOfEarth:
The TrevorOfEarth class represents a specific type of environment in a game. It extends the Environment class, indicating that it inherits behavior and properties from the parent class.
The spawn() method is overridden from the parent class Environment and is responsible for spawning an enemy at a given location in the game map. The method takes the location and map parameters, representing the location where the enemy should spawn and the game map in which the enemy is spawned, respectively.

Enemy spawning areas:
North: TrollKnight
South: GlintstoneSorcerer

Graveyard:
Enemy spawning areas:
North-west: SkeletalMilitiaman
South-west: HeavySkeletalSwordsman
North-East: SkeletalBandit
South-East: SkeletonMage

GustOfWind:
Enemy spawning areas:
North-west: DireWolf
South-west: LoneWolf
North-East: GiantDog
South-East: RottenStray

PuddleOfWater:
Enemy spawning areas:
North-west: GiantBat
South-west: GiantCrab
North-East: GiantCrayfish
South-East: GiantDragonfly

Gameactors:

Enemies:

EarthEnemy (abstract class):
The code represents an abstract class called EarthEnemy which extends the Enemy class and implements the Resettable interface. This class is designed to represent enemies in a game that belong to the Earth type. The rationale behind this design is to provide a common structure and behavior for all Earth enemies while allowing customization through subclassing. If the player kills the earth enemy, the player immediately receives max HP.

GlintstoneSorcerer:
GlintstoneSorcerer class extends the EarthEnemy class. This class represents a specific type of enemy in a game, the Glintstone Sorcerer. The Glintstone Sorcerer has a specific name, health points, attack power, and defense power. It has an intrinsic weapon that devastates other actors. It spawns on the south side of the map.

TrollKnight:
TrollKnight class extends the EarthEnemy class. This class represents a specific type of enemy in a game, the TrollKnight. The TrollKnight has a specific name, health points, attack power, and defense power. It has an intrinsic weapon that devastates other actors. It spawns on the north side of the map.

FireEnemy (abstract class):
The FireEnemy class is designed as an abstract class that extends the Enemy class and implements the Resettable interface. It represents a specific type of enemy in a game. The constructor of FireEnemy initializes the attributes inherited from the Enemy class and adds specific capabilities to the enemy.

FireMonk:
FireMonk extends another class called FireEnemy. The FireMonk class is specifically designed to represent a type of enemy called Fire Monk in a game or simulation. It has a constructor that initializes the properties of the FireMonk enemy, such as its name, symbol, health, damage, and defense values. Additionally, it overrides a method called getIntrinsicWeapon() to provide the intrinsic weapon of the FireMonk enemy, which burns other actors. It spawns on the north side of the map.

BlackflameMonk:
BlackflameMonk extends another class called FireEnemy. The BlackflameMonk class is specifically designed to represent a type of enemy called Fire Monk in a game or simulation. It has a constructor that initializes the properties of the BlackflameMonk enemy, such as its name, symbol, health, damage, and defense values. Additionally, it overrides a method called getIntrinsicWeapon() to provide the intrinsic weapon of the BlackflameMonk enemy, which burns other actors. It spawns on the south side of the map.

SkeletalMilitiaman:
The SkeletalMilitiaman class is a subclass of the Skeleton abstract class. It turns into a PileOfBones when it dies, can be be revived back. It overrides the revive method from the parent class to return a new instance of itself, and it adds a Scimitar weapon to its inventory in the constructor. Revivable interface is implemented here. It spawns on the north-west side of the map.

SkeletonMage:
The SkeletonMage class is a subclass of the Skeleton abstract class. It turns into a PileOfBones when it dies, can be be revived back. It overrides the revive method from the parent class to return a new instance of itself, and it adds a Grossmesser weapon to its inventory in the constructor. Revivable interface is implemented here. It spawns on the south-east side of the map.

DireWolf:
The DireWolf class is a subclass of WindEnemy and represents a type of enemy in a game. The DireWolf has an intrinsic weapon bite which allows it to deal damage to its enemies. It spawns on the north-west side of the map.

RottenStray:
The RottenStray class is a subclass of WindEnemy and represents a type of enemy in a game. The RottenStray has an intrinsic weapon bite which allows it to deal damage to its enemies. It spawns on the south-east side of the map.

GiantDragonfly:
A class named GiantDragonfly is made that extends the OceanEnemy class that can perform a slam attack. The getIntrinsicWeapon method of GiantCrab returns an instance of IntrinsicWeapon class, which is a weapon that is built into the creature's anatomy and does not require the creature to wield a separate object. It spawns on the south-east side of the map.

GiantBat:
A class named GiantBat is made that extends the OceanEnemy class that can perform a slam attack. The getIntrinsicWeapon method of GiantCrab returns an instance of IntrinsicWeapon class, which is a weapon that is built into the creature's anatomy and does not require the creature to wield a separate object. It spawns on the north-west side of the map.

DeathRuneDropper: (Interface)
The DeathRuneDropper interface is designed to represent an enemy that drops a death rune when defeated in a game. This interface defines a single method getDeathRune() that returns the death rune dropped by the enemy. By implementing this interface, the enemies can drop death runes when they are defeated in the game.

Contribution log

FIT2099 Assignment Contribution Log

A	B	C	D	E	F	G	H
Task/Contribution(~30 words)	**Contribution type**	**Planning Date**	**Contributor**	**Status**	**Actual Completion Date**	**Extra notes**	
Splitting tasks	Discussion	06/05/2023	EVERYONE	DONE	06/05/2023	Tanul - REQ1 maps, Satoshi - REQ1 cliff, Aditti - REQ2	
more task splitting and discussion	Discussion	12/05/2023	EVERYONE	DONE	12/05/2023	Tanul - further REQ1 maps, Satoshi - REQ3, Aditti - REQ4 astrologer part	
implemention of REQ4 Ally and invader	Brainstorm	14/05/2023	EVERYONE	DONE		Had a meeting with everyone, discussed summon sign, ally and invaders, finished basic implmentation together, Tanul took up reset, Satoshi took up actions, Aditti took up REQ5	
Worked on REQ4 and REQ5	Code comment	16/05/2023	EVERYONE	DONE	16/05/2023	finished implementation of Ally/invader, added more functionality in REQ5 - fire environment	
UML - REQ1,2,4,5 and javadoc	UML diagram	21/05/2023	ADITTI GUPTA	DONE	21/05/2023	Modified the UML diagram as per requirements and added in the javadoc code comment	
implementaion of REQ2	Code comment	06/05/2023	ADITTI GUPTA	DONE	08/05/2023	implemented new enemies and new environments as per REQ2	
implementaion of REQ4 - astrologer	Code comment	12/05/2023	ADITTI GUPTA	DONE	12/05/2023	implemented astrologer from REQ4 and helped with ally/invaders	
implementation of REQ5	Code comment	14/05/2023	ADITTI GUPTA	DONE	15/05/2023	implemented the creative requirment REQ5 - added more enemies and environments and split the map into 4 parts	
implementation of REQ4 RESET AND REST	Code comment	14/05/2023	TANUL GUPTA	DONE	15/05/2023	Implemented extra features to make sure ALLY and Invader are not removed on rest	
Implementation Of REQ1 GOLDENFOGDOOR	Code comment	06/05/2023	TANUL GUPTA	DONE	10/05/2023	Completed the implementation of Fog door and optional parts for the site of lost grace	
Made changes based on assignment 2 feedback	Code comment	18/05/2023	TANUL GUPTA	DONE	18/05/2023		
Implemented Cliff	Code comment	06/05/2023	SATOSHI KASHIMA	DONE	06/05/2023		
Implemented Golden Rune	Code comment	13/05/2023	SATOSHI KASHIMA	DONE	13/05/2023		
Implemented FingerReaderEnia	Code comment	13/05/2023	SATOSHI KASHIMA	DONE	13/05/2023		
Implemented part of Invader and Ally, and updated AttackBehaviour, FollowBehaviour	Code comment	20/05/2023	SATOSHI KASHIMA	DONE	21/05/2023		
Implemented RuneManager	Code comment	20/05/23	TANUL GUPTA	DONE	20/05/2023	Implemented and updated all teh code to work with PlayerRuneManager	
UML for A3 req3	UML diagram	21/05/2023	TANUL GUPTA	DONE	21/05/2023	Completed the UML Diagram	