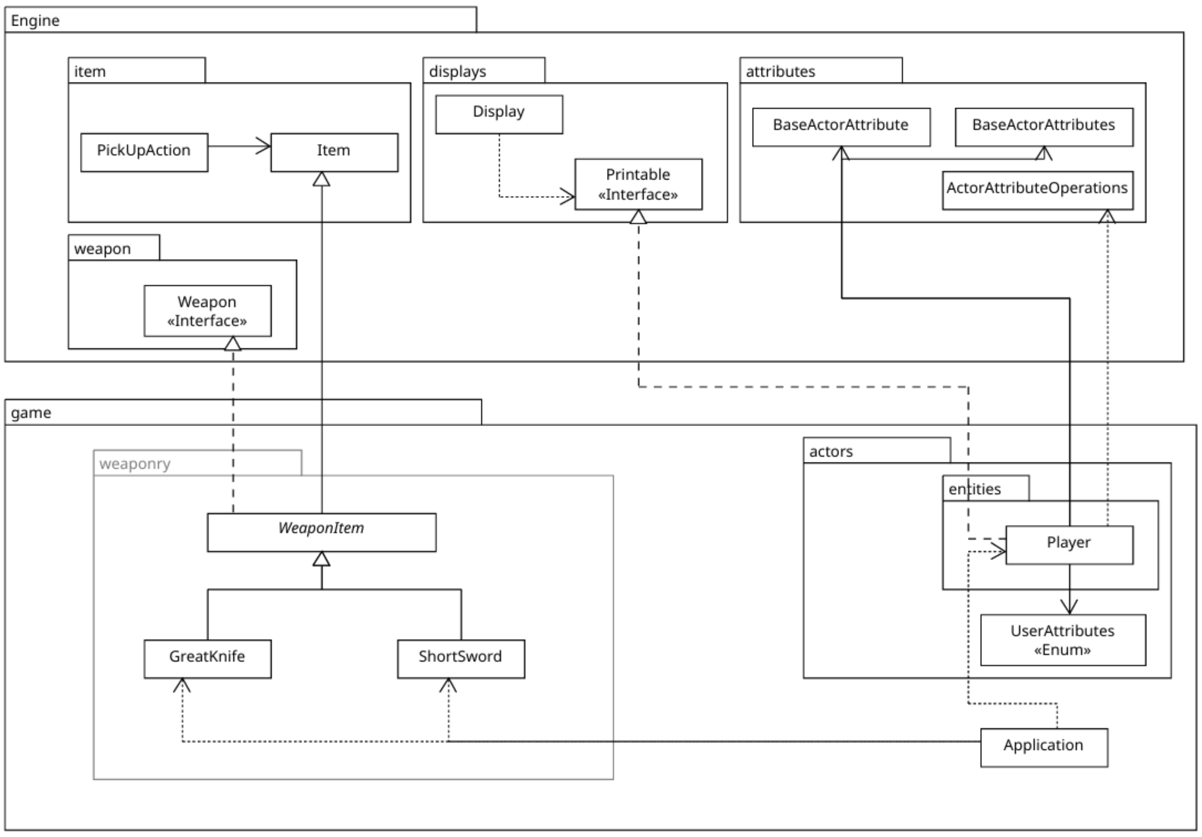
2099 Assignment 1

Dylan Matthew Quah Kwang Yung

34898573

Requirement 1:



Design Goal:

The existing concrete classes should be changed to adhere to SOLID principles by implementing abstract classes. Additionally, promoting abstraction and extensibility will enable the system to create new weapon items without altering the base template, using an abstract class.

Design Concept:

The weapon items have a required strength attribute that the player must have to pick them up. The player has an inventory capable of storing various types of weapons, which are extensions of abstract classes. The player will be able to gain stronger levels of the attributes (See REQ2), which will allow them to pick up better weapons.

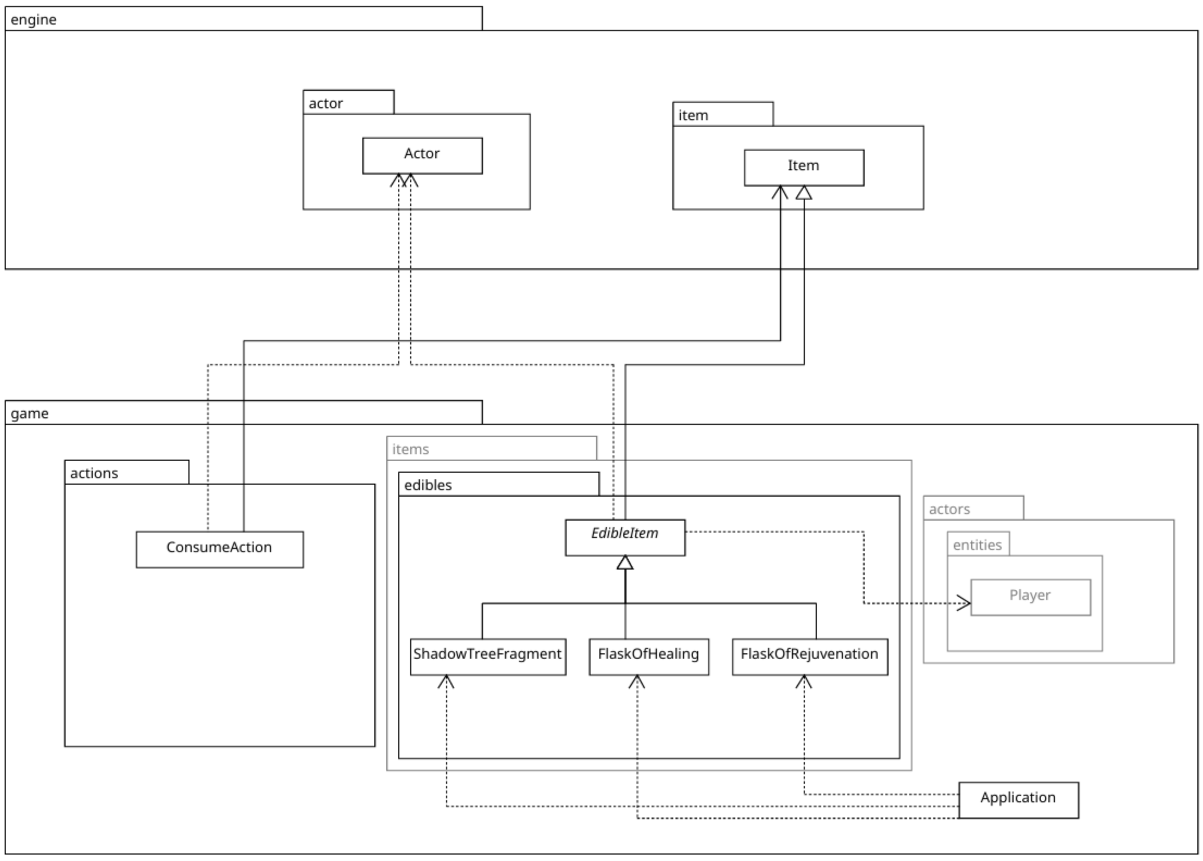
Design Rationale:

The WeaponItem abstract class serves as the foundation for all weapon items, ensuring adherence to the Single Responsibility Principle (SRP) by encapsulating the core logic and attributes related to weapons, such as damage, hit rate, and the ability to toggle portability based on the actor's strength. The GreatKnife and ShortSword classes extend WeaponItem, inheriting its functionality unless specific methods are overridden. This supports the Open/Closed Principle (OCP) by allowing new weapons to be added through extension without modifying the existing class structure. Additionally, the DRY principle is maintained as the subclasses leverage the shared logic within WeaponItem and Item.

The design promotes reduced coupling between the player and weapon classes, aligning with the Dependency Inversion Principle (DIP). Players interact with weapon subclasses through the WeaponItem interface rather than depending directly on the concrete weapon classes, enhancing flexibility. This also adheres to the Interface Segregation Principle (ISP) by ensuring the player class only interacts with relevant methods from WeaponItem, without requiring knowledge of specific weapon behaviors.

Furthermore, the checkStrength() method in WeaponItem checks the actor's strength attribute to determine whether the weapon can be made portable, further decoupling the player class from weapon-specific logic. The GreatKnife and ShortSword classes can override this method while still complying with the Liskov Substitution Principle (LSP).

Requirement 2:



Design Goal:

The new EdibleItem classes, such as FlaskOfHealing and FlaskOfRejuvenation, should be refactored to adhere to SOLID principles by implementing an abstract class. This approach enhances abstraction and extensibility, allowing the system to easily accommodate the creation of various EdibleItem types, including non-flask items like ShadowTreeFragment.

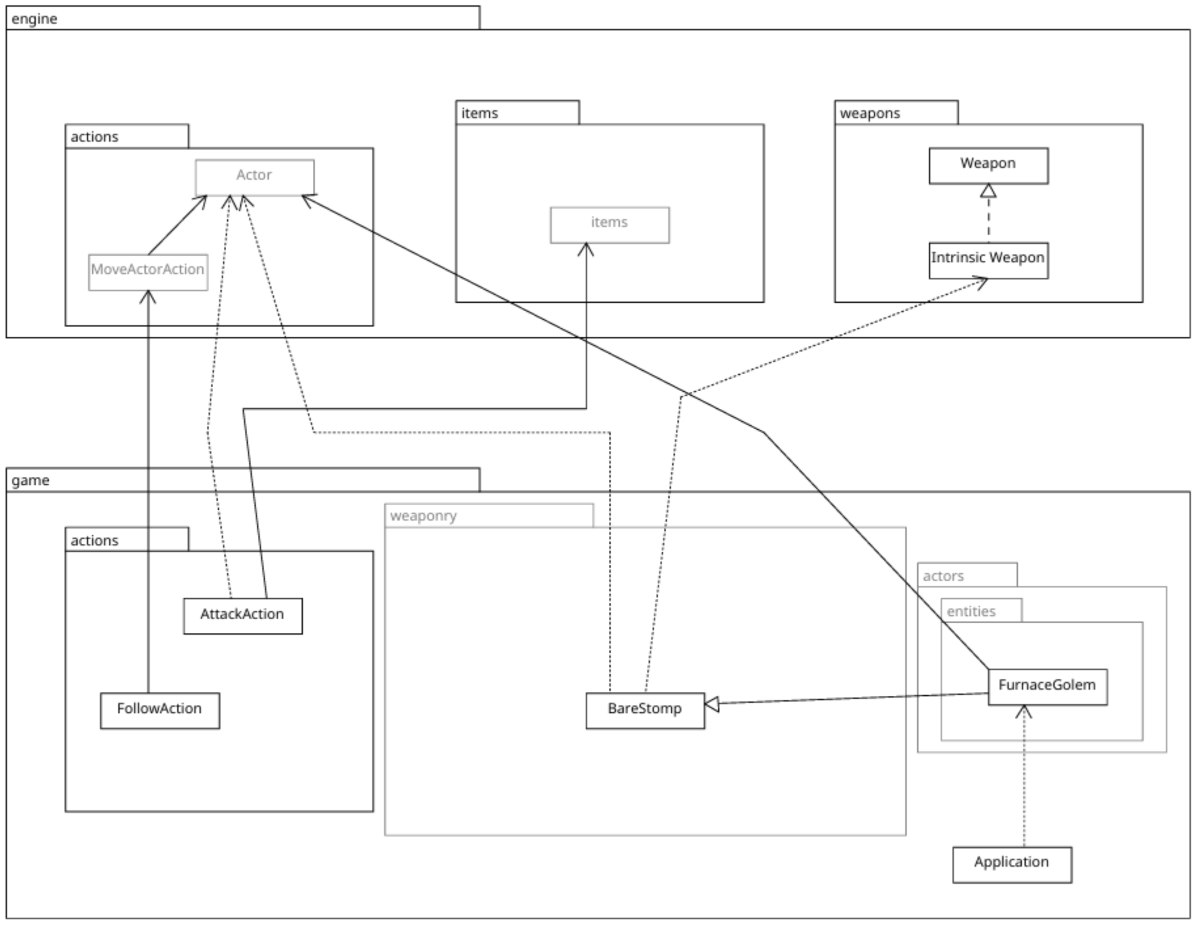
Design Concept:

Edible items are used by the player, each with different effects. The player's inventory can store various types of consumable items, which will inherit from the Item class provided by the engine. The player's attributes can be modified by the edible items, allowing them to pick up weapons that depend on the new strength attribute. The diagram represents an object-oriented system for different consumable items, which extends from the abstract EdibleItem class. The consumption logic is centralized within the EdibleItem class, allowing child classes, such as ShadowTreeFragment and various flasks, to inherit consistent methods and behavior. This approach promotes loose coupling by avoiding redundant code across all concrete classes and adheres to the DRY (Don't Repeat Yourself) principle, ensuring that similar logic is shared among child classes.

Design Rationale:

The EdibleItem serves as a template for creating specific consumable items, while the ConsumeAction allows a player to consume these items. The Single Responsibility Principle (SRP) is followed by having the abstract class handle its own responsibilities. The EdibleItem is made into an abstract class, similar to the WeaponItem abstract class. The Open/Closed Principle (OCP) ensures that new consumable items can be added without modifying the base EdibleItem class, promoting DRY by allowing subclasses to inherit the consume() method and the overridden allowableActions() method. A player can consume an EdibleItem by swapping out items like the FlaskOfHealing and FlaskOfRejuvenation. This is possible because the EdibleItem contains the original consumption methods, enabling the player to consume items based on the provided functionality. The Dependency Inversion Principle (DIP) is demonstrated using the abstract class for consumable items, and the Interface Segregation Principle (ISP) is adhered to, as the EdibleItem class only depends on the necessary methods provided by the abstract class.

Requirement 3:



Design Goal:

The new behaviour classes in the FurnaceGolem class are used to implement its behaviour and logic. The StompBehaviour class specifically utilizes an AttackAction, allowing the Furnace Golem to stomp using its intrinsic weapon, BareStomp. These classes are designed to adhere to SOLID principles by implementing abstract classes within both the behaviour and action classes. Additionally, abstraction and extensibility are emphasized to ensure the system can create different behaviours for the FurnaceGolem, enabling its logic to adapt based on varying requirements.

Design Concept:

The Furnace Golem is an actor that serves as an enemy to the Tarnished. Its behaviours are StompBehaviour, FollowBehaviour, and WanderBehaviour. The Golem detects the player through these behaviours and performs corresponding actions. It is equipped with an intrinsic weapon, BareStomp, and an AttackAction that utilizes this weapon.

The diagram illustrates an object-oriented system for implementing the FurnaceGolem and its behaviours. The Golem has three behaviours—attack, follow, and wander—each implemented in the FurnaceGolem class. The Golem cycles through these behaviours, and if it detects an actor that is HOSTILE_TO_ENEMY, it attempts to deal 100 damage with the AttackAction. The AttackAction also checks if the player is conscious, displaying a death message if the player is unconscious. Additionally, the Ability.WALK_ON_FLOOR can be assigned to actors that can walk on the floor, which prevents the FurnaceGolem from entering certain areas.

Design Rationale:

The FurnaceGolem, BareStomp, AttackAction, and StompBehaviour classes each manage distinct logic, corresponding to different aspects of an actor. The FurnaceGolem is the main actor, with BareStomp serving as its weapon. The AttackAction class is utilized by the FurnaceGolem and operates based on the attributes of BareStomp. This design adheres to the Single Responsibility Principle (SRP), as each class has a specific role and functions independently without interfering with others.

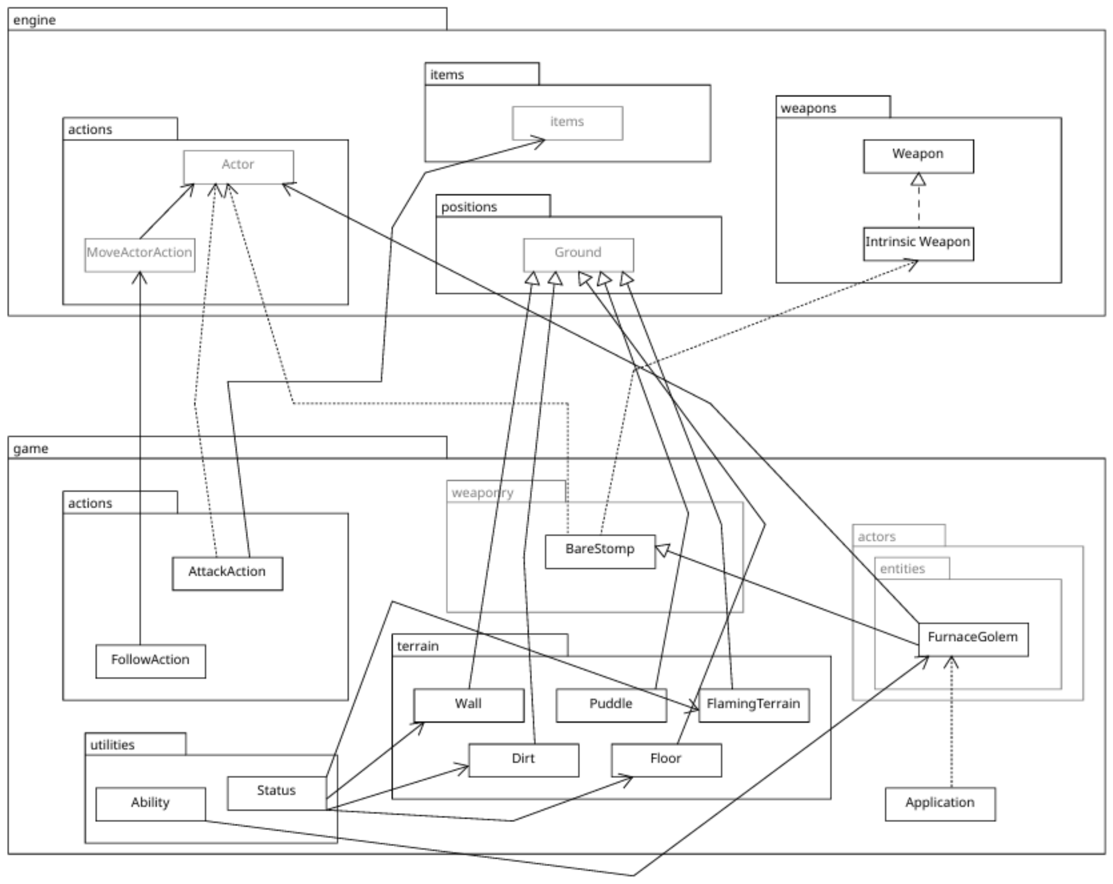
The Open/Closed Principle (OCP) is supported using a behaviour interface in the engine. This interface allows for the creation of additional specific behaviours without altering the existing code or structure of the behaviour interface.

The Liskov Substitution Principle (LSP) is followed by extending BareStomp from the InstrinsicWeapon abstract class. This enables InstrinsicWeapon and its subclasses to be substituted for BareStomp as intrinsic weapons.

The BareStomp intrinsic weapon is extended from the InstrinsicWeapon abstract class, which enforces the implementation of abstract methods by other concrete classes that follow InstrinsicWeapon, thereby adhering to the Interface Segregation Principle (ISP).

The behaviour classes that implement the Behaviour interface are abstracted from the FurnaceGolem, maintaining the Dependency Inversion Principle (DIP). The FurnaceGolem does not rely on the behaviour classes and can function even without the full implementation of a subclass that implements the Behaviour interface. Additionally, modularity is preserved as the various behaviour classes allow for the addition of new functions without affecting existing ones.

Requirement 4:



Design Goal:

The extension of functions in the FurnaceGolem and its actions should incorporate explosion logic. The game should maintain its functionality with these new changes while enabling the actors to interact with the terrain. These classes are designed to adhere to SOLID principles, even as additional logic is introduced to affect both the terrain and the FurnaceGolem's attack.

Design Concept:

The design of the Furnace Golem centers around its stomp attack, which generates an explosion that damages nearby actors and burns the surrounding ground tiles. The newly created fire tiles will persist for 5 turns, introducing a new ground type that can impact other entities. The BareStomp should be enhanced to incorporate this attack logic while maintaining its existing functionality.

The diagram illustrates an object-oriented system for implementing the FurnaceGolem's attack and terrain capabilities. The attack() method of BareStomp is extended, with the explosion logic encapsulated in a separate method called activateExplosion(), which is invoked within attack().

The activateExplosion() method in BareStomp handles the explosion logic and modifies the terrain. The FlamingTerrain class extends the Ground abstract class and implements a tick() method, which inflicts 5 damage to any player standing on a FlamingTerrain tile.

In the activateExplosion() method, surrounding tiles are checked, and those with the Status.IS_BURNING enum are converted into FlamingTerrain tiles. Tiles such as Dirt, Wall, and Floor are all eligible to be transformed into FlamingTerrain for 5 turns. Additionally, actors with the Ability.FIRE_IMMUNE trait are immune to the damage caused by the tick() method, ensuring that only non-fire-resistant actors are affected.

Design Rationale:

The Single Responsibility Principle (SRP) is demonstrated at the method level by separating the AttackAction and activateExplosion methods. This separation ensures that AttackAction doesn't handle all the logic, making maintenance easier if changes are needed in the activateExplosion functionality.

The Open/Closed Principle (OCP) is upheld as the explosion logic does not alter the AttackAction logic, allowing for additional effects to be implemented in AttackAction without modifying the existing code.

The Liskov Substitution Principle (LSP) is followed by allowing the FIRE_RESISTANT ability to be added to any actor, enabling any actor with this ability to function without requiring specific behaviour implementations.

The Interface Segregation Principle (ISP) is promoted by extending FlamingTerrain from the Ground abstract class. This ensures that only ground tiles extend from the Ground abstract class without the need for other interfaces.

The DRY (Don't Repeat Yourself) principle is applied by centralizing the burning damage logic within the FlamingTerrain tick() method. This approach allows any ground with the Status.IS_BURNING enum to include the burning capability without duplicating the damage logic in every ground subclass.