FIT2099 ASSIGNMENT 1 DESIGN DOCUMENTATION

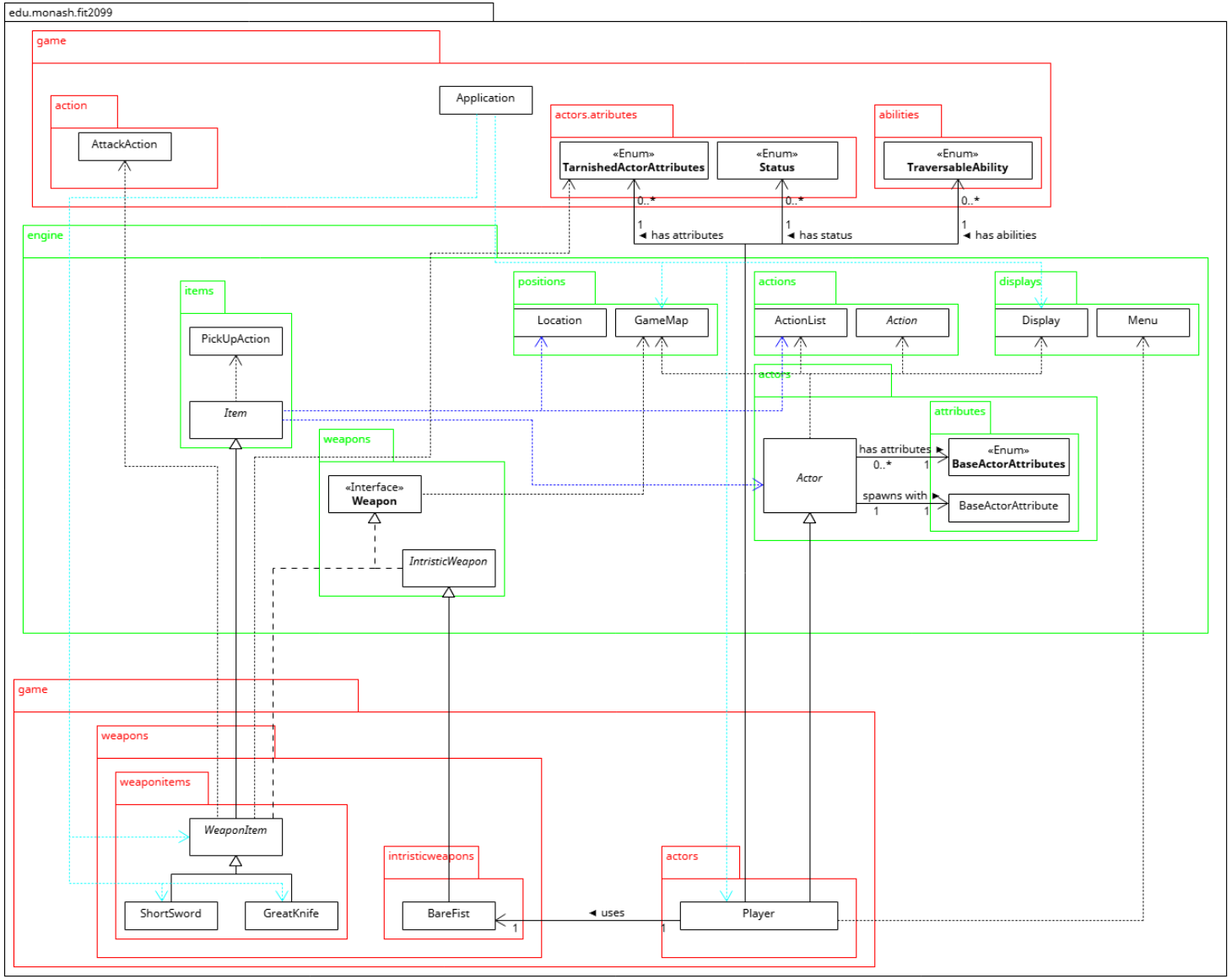
Aaron Lam Kong Yew

Design Goal

The design's objective is to decompose Elden Thing's system into smaller, more manageable classes, enhancing modularity and encapsulation. Each class will have a distinct and clear responsibility, making the system easier to understand and maintain. By following the SOLID and DRY principles, the system is designed to be easily extendable and reusable. This approach involves identifying common functionalities and abstracting them into reusable components, while also anticipating potential changes to improve code flexibility and adaptability to evolving requirements and future extensions.

Aaron Lam
33521808

Req 1 UML Class Diagram



Req 1 Design Rationale

Classes Created / Modified	Roles and Responsibility
Player (extends Actor)	Class representing a playable character that cay pick up weapon items, attack with them, and drop them Relationships: 1. Extends Actor class to include more abilities and capabilities of a generic Actor
WeaponItem (extends Item) (implements Weapon)	Abstract class representing a generic weapon Relationships: 1. Extends Item class since it can also be picked up and dropped 2. Implements Weapon interface to allow the item to be used to attack any Entity
ShortSword (extends WeaponItem)	Class representing a Short Sword item Relationships: 1. Extends WeaponItem abstract class to provide a concrete example of a weapon that can be instantiated and used.
GreatKnife (extends WeaponItem)	Class representing a Great Knife item Relationships: 1. Extends WeaponItem abstract class to provide a concrete example of a weapon that can be instantiated and used.
TarnishedActorAttributes	Enum class representing extra features besides the ones defined in BaseActorAttributes

Application	The main class to start the game.

Solution and Explanation:

Converted WeaponItem from a concrete class into an abstract class, and have ShortSword and GreatKnife classes extend from it. This is because a generic WeaponItem should not exist in the game map, as it does not have any predetermined damage or hitrate. But rather, weapons that spawn in the map have to be a concrete example of it, to be able to be picked up, used and dropped by entities. For example, the ShortSword has a predetermined damage of 100 and hit rate of 75.

Since both weapons can only be picked up when the actor has the appropriate strength to do so, an extra attribute requiredStrength was added into the WeaponItem abtract class so that all its superclasses will be assigned requiredStrength integers that the Actor must reach or exceed in order to wield them. A new Enum class TarnisehdActorAttributes was created so that the Player class can use the “STRENGTH” attribute dereived from it.

Advantages:

Single-Responsibility Design (SRP):

By converting WeaponItem into an abstract class and having ShortSword and GreatKnife extend from it, each class now has a single responsibility. WeaponItem defines the common behavior and properties of all weapons, while ShortSword and GreatKnife define specific attributes and behaviors for those particular weapons.
Example: To change the damage calculation logic, one can do it in the WeaponItem class without affecting the specific attributes of ShortSword or GreatKnife.

Open/Closed Principle (OCP):
The design allows for easy extension of new weapon types without modifying existing code. One can create new weapon classes by extending WeaponItem and defining their specific attributes.

Example: If a new weapon like LongSword needs to be added, one can simply create a new class LongSword that extends WeaponItem and define its specific attributes like damage and hit rate.

Liskov Substitution Principle (LSP):
Benefit: Any Enum class can be used wherever addAttribute is called, ensuring that the program behaves correctly.
Example: If a Player's constructor expects calls addAttribute, you can pass an instance of BaseActorAttributes or TarnishedActorAttributes without any issues.

Extensibility:
The design is easily extendable. If new weapons need to be added, one can simply create new classes that extend WeaponItem and define their specific attributes.
Example: Adding a new weapon like MagicWand would involve creating a new class MagicWand that extends WeaponItem and defining its specific attributes like damage, hit rate, and required strength.

Possible Limitation/Drawback

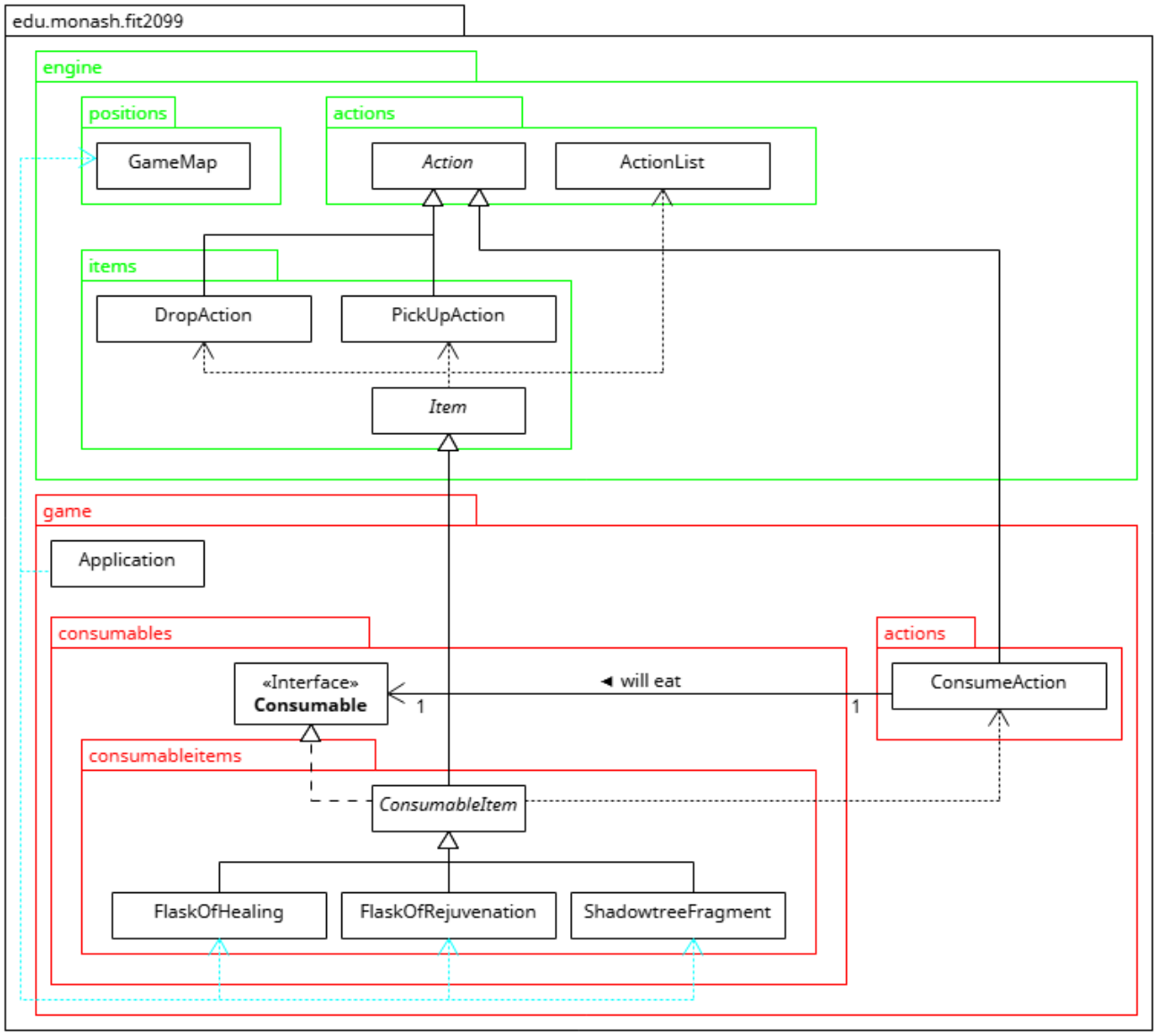
Limitation: The current design tightly couples the weapon's required strength with the weapon itself. This means that any changes to the required strength for a weapon would require modifying the weapon class directly. This can lead to a lack of flexibility if the required strength needs to be, for example adjusted dynamically based on game conditions or player attributes.

Alternative Solution

Introduce a WeaponRequirement interface or class that handles the requirements for wielding a weapon. This way, the requirements can be decoupled from the weapon itself and can be modified or extended without changing the weapon classes.

Potential Overhead of Alternative Solution: If there are multiple types of requirements (e.g., strength, agility, magic), managing these through separate classes can lead to increased overhead in terms of both development and runtime performance.

Req 2 UML Class Diagram



Req 2 Design Rationale

Classes Added / Modified	Roles and Responsibility
Consumable	Interface Representing any Game Entity that can be consumed
ConsumableItem (extends Item) (implements Consumable)	Abstract class representing an Entity that is both an Item, and can be consumed Responsibilities: 1. Extends Item, to include more attributes such as the item's charges 2. Implements Consumable so that ConsumableItem will implement the consume method
ConsumeAction (extends Action)	Class representing an action to consume any Consumable item Responsibilities: 1. Extends Action, to be able to override Action's execute and menuDescription method with a more suitable implementation
FlaskOfHealing (extends ConsumableItem)	Class representing an Item that can be consumed and will increase the Actor's current HEALTH. Responsibilities: 1. Extends ConsumableItem abstract class to provide a concrete example of a item that can be instantiated and consumed
FlaskOfRejuvenation (extends ConsumableItem)	Class representing an Item that can be consumed and will increase the Actor's current MANA. Responsibilities:

	1. Extends ConsumableItem abstract class to provide a concrete example of a item that can be instantiated and consumed
ShadowtreeFragment (extends ConsumableItem)	Class representing an Item that can be consumed and will increase the Actor's maximum HEALTH, MANA and STRENGTH Responsibilities: 1. Extends ConsumableItem abstract class to provide a concrete example of a item that can be instantiated and consumed
Application	The main class to start the game.

Solution and Explanation:

Since this requirement involves creating Items that can be eaten/consumed, it was logical to create both a Consumable interface and an abstract class ConsumableIem. The Consumable interface will only have one method, called consume, to signify that all classes that implement from it, will have a consume method (can consume some game Entity). The ConsumableItem extends from the abstract Item class so that all of ConsumableItem's superclasses will still maintain the basic properties of an Item, but now with extra attributes, such as the number of times the item can be used/consumed. The ConsumeAction class is also created and extends from Action since it is also an action, having all its properties such as a description, and a way of executing itself. The ConsumeAction class created will then become a dependency inside ConsumableItem's allowableActions method. These means that all Items that can be consumed will have a ConsumeAction to perform on. The 3 superclasses of ConsumableIem will be able to benefit from the methods already predefined inside ConsumableIem.

Advantages:

Liskov Substitution Principle (LSP):

Any subclass of ConsumableItem can be used wherever ConsumableItem is expected, ensuring that the program behaves correctly. For example, if a method expects a ConsumableItem, one can simply pass an instance of FlaskOfHealing or ShadowtreeFragment, as well as other items with the same reference class of ConsumableItem without any issues. This also embraces the Polymorphism concept, as different dynamic class types are joint together by the same reference class.

Interface Segregation Principle (ISP):

By defining specific method in Consumable, one can ensure that future classes only implement the methods they need. For instance, if a new type of consumable requires additional methods, one can define them in the subclass without affecting other existing consumable types.

DRY Principle:

The ConsumableItem abstract class encapsulates common properties and behaviors of consumable items, such as having "*actions.add(new ConsumeAction(this))*" and as such, reducing code duplication.

Extensibility:

If new consumable items need to be added, one can simply create new classes that extend ConsumableItem and define their usual attributes. Adding a new consumable item like BubbleMilkTea would only involve creating a new class BubbleMilkTea that extends ConsumableItem and defining its specific attributes like health or mana restoration.

Possible Limitation/Drawback:

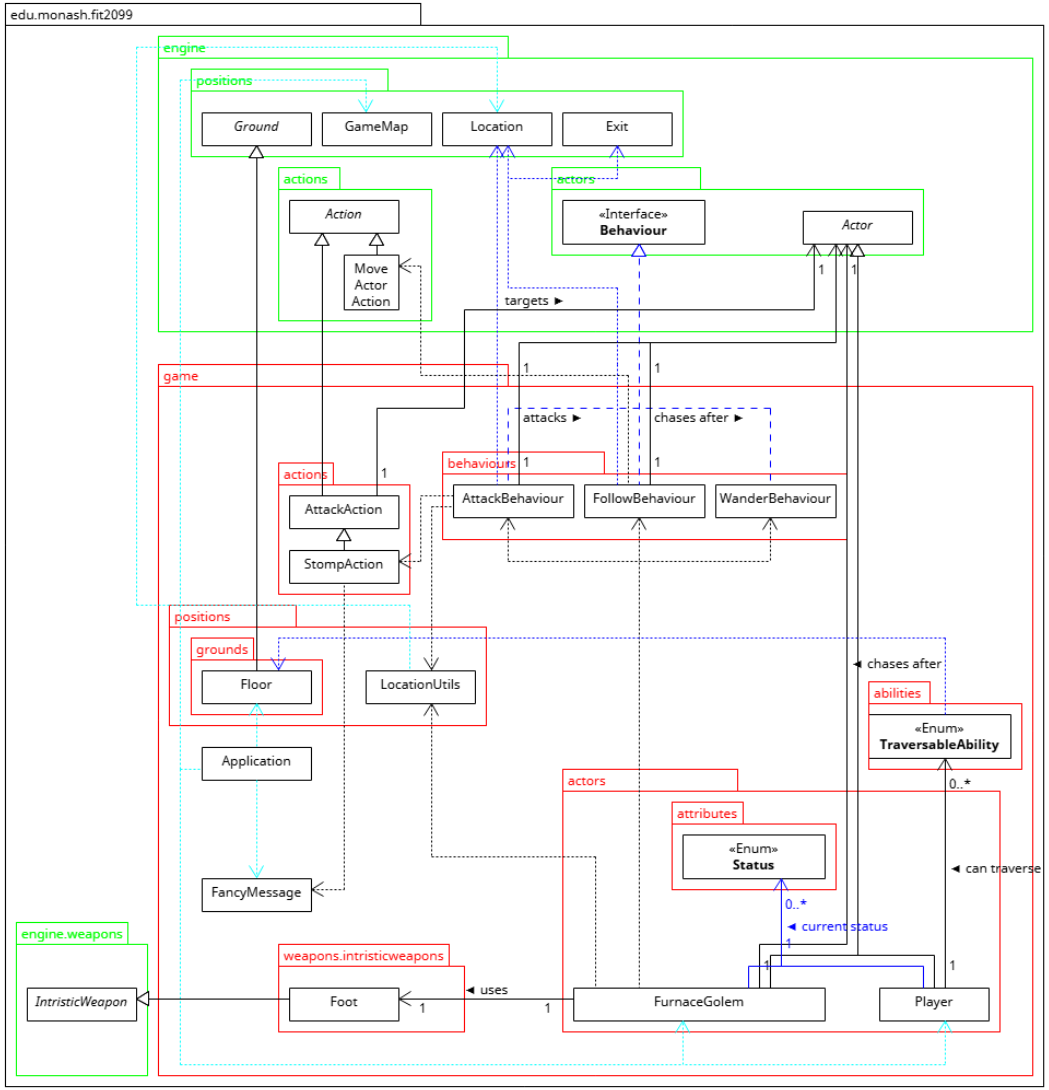
The current design tightly couples the consumable logic with the item itself. This means that any changes to the ConsumeAction class would require modifying the food item's class directly. This can lead to a lack of flexibility if the consumable behavior needs to be adjusted dynamically based on game conditions or player attributes.

Alternative Solution:

Potentially introduce a ConsumableBehavior interface or class that handles the behavior for consuming an item. This way, the behavior can be decoupled from the item itself and can be modified or extended without changing the item classes.

Drawback of Increased Complexity: Introducing a ConsumableBehavior interface adds an additional layer of complexity to the codebase. Developers need to understand and manage the new interface and its implementations, which requires more time and energy to understand each layer of implementation, from *ConsumableBehaviour -> ConsumeAction -> Item's consume*.

Req 3 UML Class Diagram



Req 3 Design Rationale

Class Added / Modified	Roles and Responsibility
FurnaceGolem (extends Actor)	Class representing an NPC Actor that is hostile towards the player Responsibilities: 1. Extends abstract Actor class to utilize allowableActions method and playTurn method from it
FollowBehaviour (implements Behaviour)	Class representing a behaviour whereby the Actor follows a target Responsibilities: 1. Implements the Behaviour interface to be implement the getAction method and return MoveActorAction to move the Actor
AttackBehaviour (implements Behaviour)	Class representing a behaviour whereby the Actor attacks a target Responsibilities: 1. Implements the Behaviour interface to be implement the getAction method and return StompAction to attack the target
StompAction (extends AttackAction)	Class representing an action to perform specific attack style on a target Responsibilities: 1. Extends AttackAction, to be able to override AttackAction's execute method with a more complicated implementation
Foot (extends IntrinsicWeapon)	Class representing an example of one intrinsic weapon, whereby each Actor has a maximum of only one intrinsic weapon Responsibilities:

	1. Extends IntrinsicWeapon to carry the representation of a weapon for an unarmed Actor (e.g. fists, claws, etc.)
Player (extends Actor)	Class representing a playable character that can be attacked by a Hostile enemy, e.g. FurnaceGolem Relationships: 1. Extends Actor class to include more abilities and capabilities of a generic Actor
LocationUtils	A utility class that provides helper methods, specifically made to handle Location related functions / calculations, such as checking if two entities are adjacent
TraversableAbility	Enum class that represents whether a non-stationary actor can traverse a certain type of ground.
Floor (extends Ground)	Class that represents the internal grounds of the shack the Player spawns in. Only Actors will specific attributes can stand on them Responsibilities: 1. Extends Ground abstract class, to be able to check if an actor can enter the floor. The actor must have TraversableAbility.FLOORTRAVERSABLE ability.
Wall (extends Ground)	Class that represents the outline grounds of the shack the Player spawns in. Only Actors will specific attributes can stand (climb) on them Responsibilities:

	1. Extends Ground abstract class, to be able to check if an actor can enter the wall. The actor must have TraversableAbility.WALLTRAVERSABLE ability.
FancyMessage	Display class representing fancy messages that are used, to print the game title and the end of game, by calling its static methods
Application	The main class to start the game.

Solution and Explanation:

For this requirement, the Furnace Golem must be able to detect when the player is adjacent to it, and if yes, to attack the player. If the player attempts to run away, keep following the player until one of them loses consciousness, or the player retreats safely into the shack. Here is a mechanical breakdown of this complex implementation, explaining how the (existing and new) classes will interact to deliver the required functionality:

1. Detect player
    a. As the Furnace Golem wanders around (with WanderBehaviour) every tick, the LocationUtils.*isAdjacent* helper method is invoked to scan the Golem's surrounding and check if a player (or other Entities the Golem is hostile towards) is within its primary surroundings.
    b. This ensures the Golem is able to sense when the player is within reach, whilst maintaining its original (lowest priority behaviour) WanderBehaviour.
2. Stomp on player
    a. Once the Golem detects the player, it's priority shifts from WanderBehaviour to AttackBehaviour, whereby the StompAction will be invoked by the Golem's AttackBehaviour. Using different integers as keys will contribute to

the priority of an NPC's behaviour, wheby the lower the key number, the higher its priority. As such, the priority for the Furnace Golem is currently AttackBehaviour > FollowBehaviour > WanderBehaviour.

b. Using the new Feet weapon as the Furnace Golem's intrinsic weapon, it will attempt to damage the player for every tick the player remains in its *adjacent* surroundings. Since this requirement states the that the Golem will attack the player but doesn't state that it will pick up any weapon, this Feet weapon class has been created and added as the Golem's intrinsic weapon. This maintains the original design of the game, whereby all Game Entities must wield (or intrinsically have) a weapon in order to deal damage.
c. If the player dies in this battle, *FancyMessage*.*printYouDied*() will be invoked inside the StompAction class to signify the end of the game. Setting *printYouDied* method as a static method allows it to be called easily, without requiring to create an instance of a FancyMessage class each time its methods are required

3. Follow player
    a. If the player attempt to run away, the Golem's AttackBehaviour priority is removed and its priority will shift towards FollowBehaviour. This behaviour logic still holds.
    b. FollowBehaviour will calculate the shortest path to take and will cause the Golem to chase the player each tick
    c. Since the player has the ***WALLTRAVERSABLE*** attribute and the Golem does not, the player can safely retreat into his shack, while the Golem has no choice but to wait outside, while not loosing the FollowBehaviour priority
    d. Once the player exits the Floor "(_)" ground, step 2 will repeat until one of them dies.

Advantages:

Single Responsibility Principle (SRP):

Each class and behaviour has a single responsibility. For example, FurnaceGolem stores and orders a HashMap of behaviors of the NPC, meanwhile FollowBehaviour handles chasing logic, AttackBehaviour handles attacking logic, and StompAction handles the specific attack action. In the future, if one needs to change the way the Golem attacks, he/she can modify the StompAction class without affecting the FurnaceGolem nor AttackBehaviour classes.

Open/Closed Principle (OCP):

This design allows for easy extension of new behaviors or actions without modifying existing code. One can create new behavior classes by implementing the Behaviour interface and defining their specific actions. If a new behavior like DefendBehaviour needs to be added, just create a new class DefendBehaviour that implements Behaviour and define its specific actions. In fact, since it is a Behaviour, one can adjust its priority, for example a new Whale Actor that prioritises depending rather than attacking other hostile Actors.

DRY Principle:

Both the Behaviour interface and the IntrinsicWeapon class encapsulate common properties and behaviors, reducing code duplication. To illustrate, the getAction method in Behaviour and the execute method in AttackAction are defined only once and reused by all subclasses, ensuring that the logic for behaviors and actions is consistent across all their subtypes.

Extensibility:

Adding a new behavior like DefendBehaviour is very simple, as it would only involve creating a new class DefendBehaviour that implements Behaviour.

Possible Limitation/Drawback:

The current design tightly couples the Behavior logic with the FurnaceGolem class. This means that any changes to the Behavior logic would require modifying the FurnaceGolem class directly. This can lead to a lack of flexibility if the behavior logic needs to be adjusted dynamically based on game conditions or player attributes.

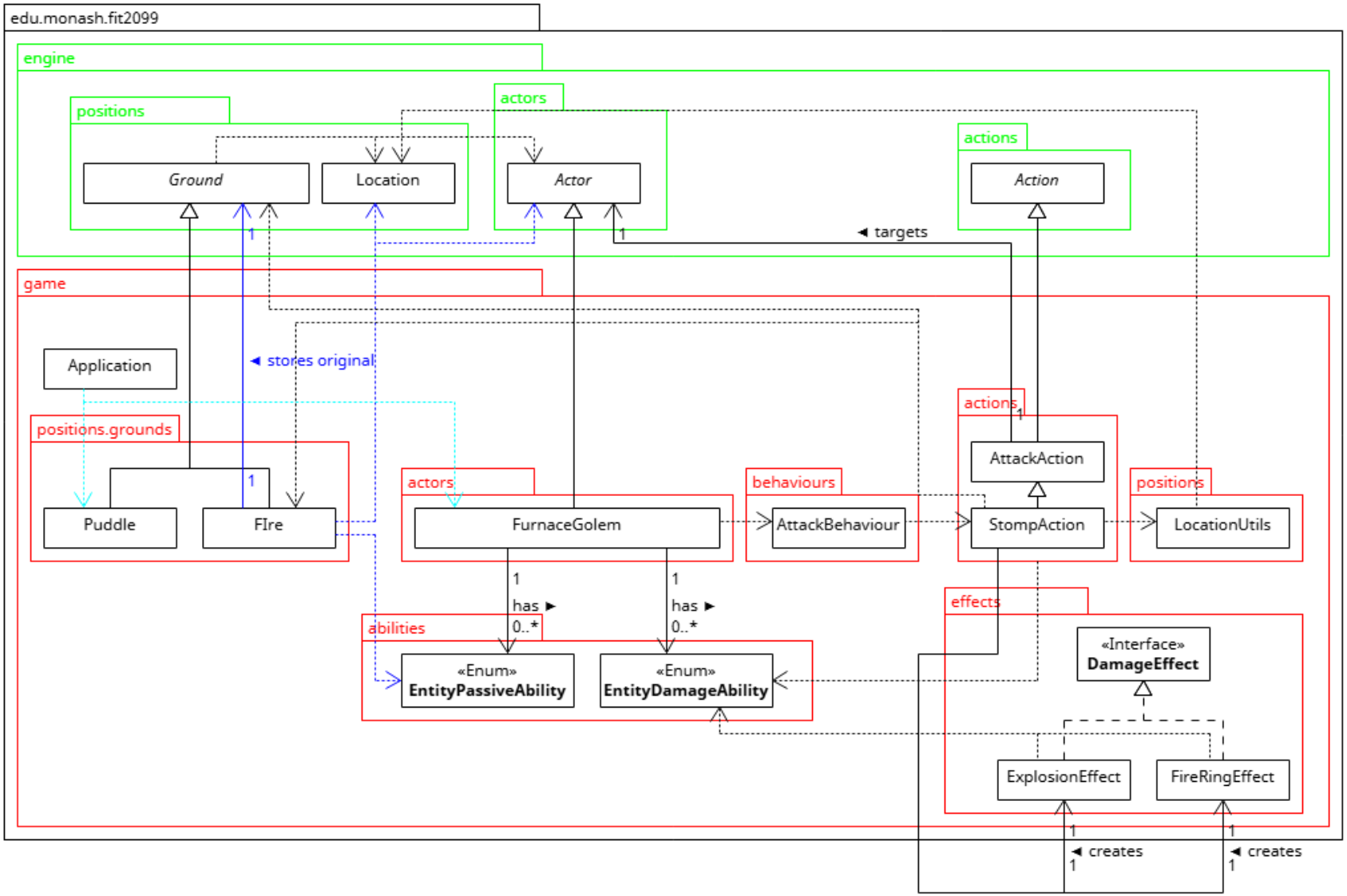
Alternative Solution:

Combining StompAction and AttackAction into a singular class.

This will reduce code complexity as it indicates that all forms of Attack will be handled solely by AttackAction.

Drawback of OCP violation: Since AttackAction is shared by multiple Actors, it is impossible to execute different attack patterns for each Actor. For example, some actors can deal critical damage ocassionally, which requires the existing AttackAction to be modified to cater specifically just for them.

Req 4 UML Class Diagram:



Req 4 Design Rationale

Class Created / Modified	Roles and Responsibility
Puddle (extends Ground)	Class representing a Puddle ground that cannot be set on fire Responsibility: 1. Extends Ground abstract class to implement its method, canActorEnter
Fire (extends Ground)	Class representing a Fire ground that causes 5 damage for 5 ticks Responsibility: 1. Extends Ground abstract class to implement its methods, canActorEnter and tick
FurnaceGolem (extends Actor)	Class representing an NPC Actor that is hostile towards the player, and has certain passive and damage abiliities Responsibilities: 1. Extends abstract Actor class to utilize allowableActions method and playTurn method from it
EntityPassiveAbility	Enum class representing extra non-damaging abilities that an entity can have.
EntityDamageAbility	Enum class representing extra damaging abilities that an entity can have, besides their AttackAction (or its subclasses)
StompAction (extends Action)	Class representing an action to perform specific attack style on a target, and may include extra damage effects

	Responsibilities: 1. Extends AttackAction, to be able to override AttackAction's execute method with a more complicated implementation, *such as dealing explosion damage and setting the surrounding ground on fire*
LocationUtils	A utility class that provides helper methods, specifically made to handle Location related functions / calculations, such as retrieving a list of locations that are adjacent to any given current location.
DamageEffect	An interface that represents an effect that deals damage to actors. Requires the actor to have the capability to deal damage.
ExplosionEffect (implements DamageEffect)	A class that represents an effect that deals ExplosionEffect damage to actors. Responsibilities: 1. Implements DamageEffect to apply the extra explosion damage to surrounding actors.
FireRingEffect (implements DamageEffect)	A class that represents an effect that deals damage to actors, by changing the ground to fire. Responsibilities: 1. Implements DamageEffect to set the surrounding ground on fire.
Application	The main class to start the game.

Solution and Explanation:

Since the Furnace Golem attacks now deal explosion damage and a ring of fire, the Stomp Action class was further modified to include these extra "side-attacks" that may occur each time StompAction is called, under the DamageEffect Interface. These "side-attacks" can be carried out by any Actors that have ***EXPLOSION*** and ***FIRE_RING*** attributes from the EntityDamageAbility Enum, by checking for them inside both DamageEffect attributes of the StompAction class. This allows flexibility of attack combinations in the current design, for example a new Dragon Actor may have the ***FIRE_RING*** attribute, but not ***EXPLOSION,*** so in such cases, these attributes must have the functionality to be able to occur both dependently or independently from one another. Since ExplosionEffect and FireEffect are two separate classes that can be called independantly, this increases the flexibility of an Actor having multiple combinations of attack effects.

Additionally, when the ***FIRE_RING*** attribute occurs (every 10%), it changes the surrounding Location of the Furnace Golem from their original ground into Fire Ground. Thus, creating a originalGround object as an attribute of the Fire class was appropriate, so that the Game Map restores the original Location set on Fire after the Fire completes its ticks and extinguishes. This also ensures that the Golem is unable to walk through walls or floors set on fire, as the Fire class copies and stores the permeability of its original ground using the *canActorEnter* method.

Next, since a Puddle GameEntity and the Furnace Golem GameEntity cannot catch on fire, it was logical to create a new Enum class EntityPassiveAbility. Thus, both entities will have FIRE_RESISTANCE as part of their *capabilitySet*. Depending on whether the entity is a Ground or Actor, the Puddle object will not be overridden by the Fire (for Ground), and the Furnace Golem will not suffer burn damage each tick (for Actors). This ensures the requirement is being followed, while catering for future GameEntities that may also possess FIRE_RESISTANCE hence ensuring high cohesion between classes.

Advantages:

Liskov Substitution Principle (LSP):

Any subclass of Ground or Actor can be used wherever those types are expected, ensuring that the program behaves correctly. For example, say a method expects a Ground type to check whether it can be burnt, one can pass an instance of Puddle, Fire or any other Ground subclasses without any issues.

DRY Principle

The Ground abstract class encapsulates common properties and behaviors of grounds, reducing the need for code duplication. The canActorEnter method is defined once in Ground and reused by all subclasses, ensuring that the logic for entering a ground is consistent across all ground types. If needed, these subclasses may @Override the predefined canActorEnter method to cater to their specific functionality.

Extensibility

If new ground types or abilities need to be added, one can simply create new classes that extend Ground or implement new abilities in the respective enums.

To elucidate, adding a new ground type like Lava would involve only creating a new class Lava that extends Ground and defining its specific behaviors. To create more attack effects, one would only need to add them inside the EntityDamageAbility and implement their damage effects when invoked.

In addition, creating new damage effects become very simple as one only needs to create a new class that implements DamageEffect interface, and then call them from any Action class if he/she wishes to.

Possible Limitation/Drawback:

Limitation: The current design slighty couples the behavior logic with the StompAction class. This means that any changes to the AttackBehavior logic would require modifying the StompAction class directly. This could lead to a lack of flexibility if the behavior logic needs to be adjusted dynamically based on game conditions or player attributes.

Alternative Solution:

Making ***EXPLOSION*** and ***FIRE_RING*** belong (as attributes) of the Foot Weapon, rather than of the Furnace Golem. This allows future Actors that equip the same Foot intrinsic weapon to automatically equip both special attacks. This implementation may reduce code complexity, as it becomes simple to recognize that all Foot Weapons created will come with these 2 attributes.

Drawbacks: It will not be possible for future Actors to deal explosion damage and fire ring damage, unless they first spawn with Foot as their intrinsic weapon. This may tighten coupling between classes, which in turn may reduce the flexibility and combinations of attacks that can be done by each Actor. It is more appropriate that these attributes are not withheld by their weapons, but rather are free to be implemented by a variety of Actors.