MONASH University

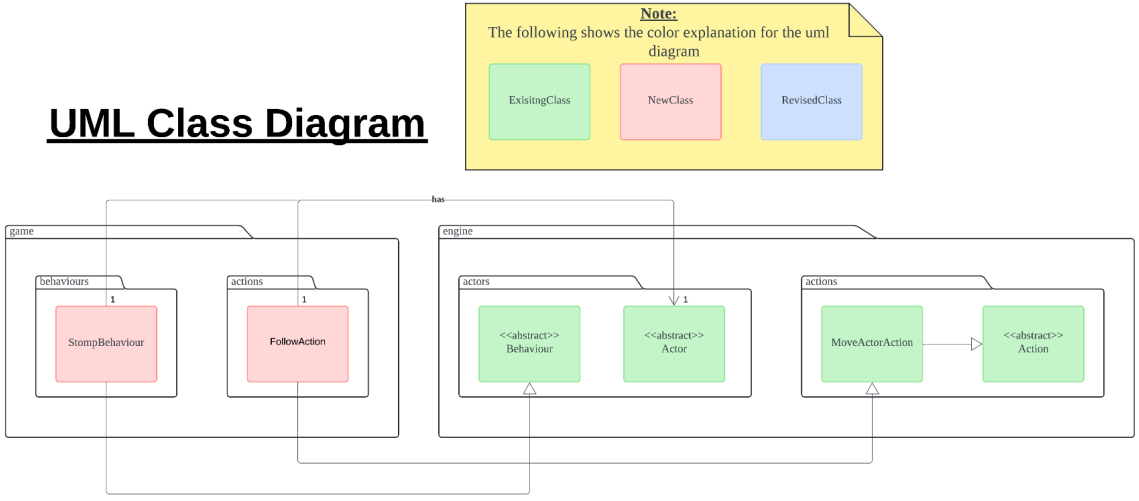
Assignment 2 Design Rationale

Prerequisite: Modification to A1

- Created StompBehaviour class and FollowAction is created for A2 from A1 in order to:
    a. Make sure that AttackBehaviour returns only AttackAction and only handles normal basic attacks instead of StompAction
    b. FollowAction which is returned by FollowBehaviour

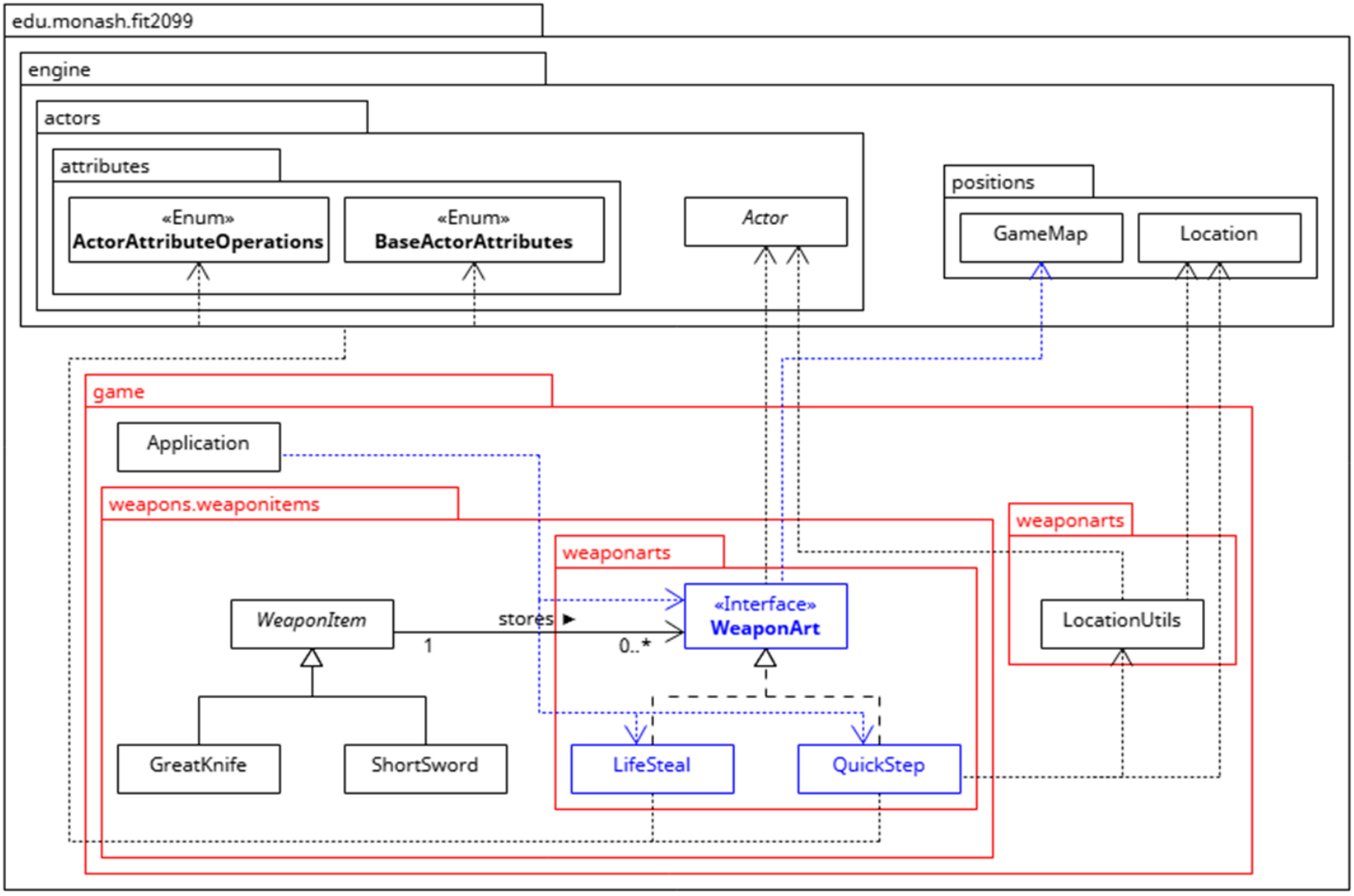
A1 changes

UML Class Diagram



MONASH University

Req 1 UML Class Diagram



MONASH University

Req 1 Design Rationale

Classes Created / Modified	Roles and Responsibility
WeaponArt	Interface representing special abilities that a WeaponItem may possess (1 or more)
LifeSteal (implements WeaponArt)	Class representing a life steal ability that can be equipped by a concrete WeaponItem Responsibility: 1. Implements WeaponArt to provide a weapon with self-healing ability
QuickStep (implements WeaponArt)	Class representing a “quick-step” ability that can be equipped by a concrete WeaponItem Responsibility: 1. Implements WeaponArt to provide a weapon with a 1-tile movement ability

MONASH University

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
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.
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.

LocationUtils	A utility class that provides helper methods, specifically made to handle Location related functions / calculations, such as generating a random Location to move to.
Application	The main class to start the game.

Solution and Explanation:

Created a new interface, WeaponArt that provides methods such as execute, and getRequiredMana. Future subclasses that implement this interface, such as LifeSteal and QuickStep will have to implement these methods too, thus ensuring the subclass fulfilling the WeaponArt interface must also fulfil its abstract methods. WeaponArt is made as an interface instead of an abstract class because of the potential that these weapon abilities may be used not just by weapon items, but also by other items in the future.

In WeaponItem and its subclasses, an extra parameterized constructor was created that takes in a List of WeaponArts attribute. This allows the weapon to “store” one or more weapon arts inside it. When the execute function is called, WeaponItem will loop through its WeaponArt list and execute each functionality sequentially. This ensures that any weapon instance from any weapon class can be instantiated with any of the available weapon arts. If the WeaponArt list is empty, it indicates that a weapon does not have a weapon art, hence attacking with it will simply deal normal damage of the weapon with the weapon's normal chance to hit.

An extra helper method was added in LocationUtils to generate a random adjacent Location that the attacker than move to after executing an attack using a weapon equipped with a QuickStep art. Since it was never explicitly mentioned that each weapon must only contain 1 weapon art, it was beneficial to create an attribute of weaponArts list, rather than a single weaponArt. This allows for flexibility and extensibility of a

weapon's design features, primarily allowing for more weapon arts to be infused into an existing weapon, in future implementations. For example, A flaming sword may have 2 weapon effects, LifeSteal and FireRing in the future!

Advantages:

Single Responsibility Principle (SRP):

Each class has a single responsibility. For example, LifeSteal and QuickStep each represent a specific weapon ability, and LocationUtils handles location-related calculations.

Dependency Inversion Principle (DIP):

High-level modules (WeaponItem) depend on abstractions (WeaponArt), not on concrete implementations (LifeSteal, QuickStep).

Extensibility:

The design is highly extensible, since new weapon arts can be added by simply implementing the WeaponArt interface. New weapons can then be created by extending WeaponItem and providing the desired weapon arts. This allows for easy addition of new features without modifying existing code in both the interface and the abstract class.

Preventing Connascence of Type and Connascence of Meaning:

For Requirement 1, Interfaces were used to Decouple Weapon Arts from Weapon Items. By defining an interface for WeaponArt, I decouple the implementation of weapon arts from the weapon items that use them. This prevents connascence of type because the weapon items do not need to know the specific implementation details of the weapon arts. This will then also prevent connascence of meaning because the weapon items are not tightly coupled to the specific weapon arts.

Don't Repeat Yourself Principle (DRY):

MONASH University

The *LocationUtils.getRandomSurroundingLocation* method helps in adhering to the DRY (Don't Repeat Yourself) principle by centralising the logic for finding a random surrounding location into a single reusable method in LocationUtils. This prevents the need to duplicate the same logic in multiple places throughout the codebase.

Benefits:

Reusability: The method can be called from different parts of the code whenever a random surrounding location is needed, ensuring that the same logic is applied consistently, saving time and lines of code. All logic of getRandomSurroundingLocation is modularised in LocationUtils.

Maintainability: If the logic for finding a random surrounding location needs to be updated, it can be done in one place, reducing the risk of errors and inconsistencies applied to multiple instances of the method needed to be used.

Readability: The method name clearly describes its purpose, making the code more readable and easier to understand.

Possible Limitation/Drawback

One possible limitation is that the *execute* method of the WeaponItem class might run too long if it needs to handle a large number of weapon arts. This could increase time complexity and slow down the program.

Alternative Solution

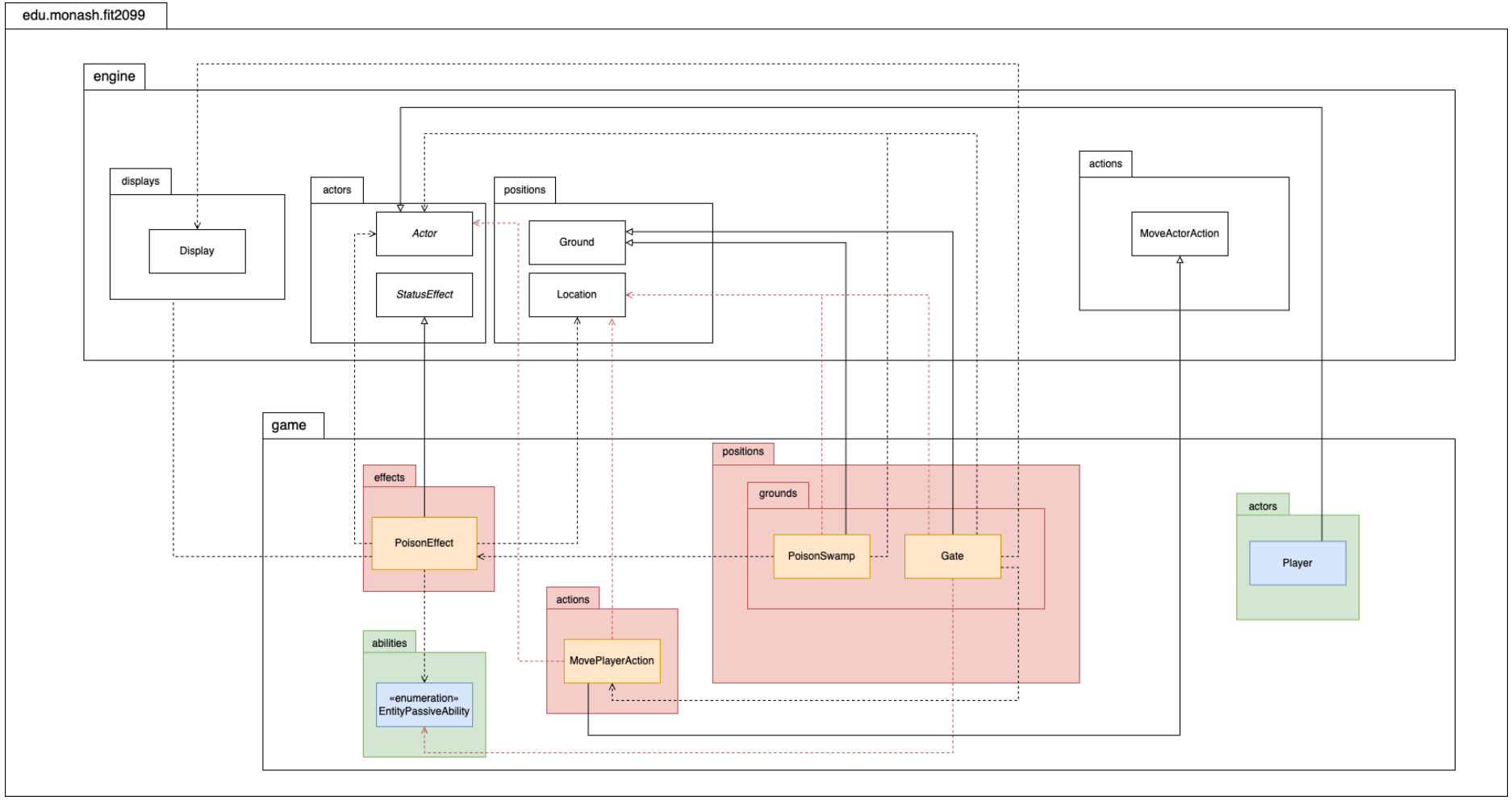
Instead of using a WeaponArt Interface and its subclasses, assign ENUM attributes to a weapon, signifying potential abilities that a weapon item may possess.

Potential Overhead of Alternative Solution: This indicates that each ENUM attribute ability of a weapon must be matched to its corresponding execute/action method somewhere in the program. This will likely increase the number of classes in the program, which could add complexity to the code and make it harder to debug and maintain.

MONASH University

Req 2: Belurat, Tower Settlement

Req 2 UML Diagram



MONASH University

Req 2 Design Rationale

Classes Created / Modified	Roles and responsibility
Application	Added new maps: i) Belurat, Tower Settlement ii) Belurat Sewers As well as new Gate grounds on all three maps to simulate travelling between maps
Gate (extends Ground)	A ground that represents a gate in the console. Responsibility: 1. Allows Player to travel between maps 2. Locations the gate transports Player to are added using the constructor or addLocation(Location) method 3. Uses MovePlayerAction to check for actors in destination location
PoisonEffect (extends StatusEffect)	A status effect that represents poison acting on non-poison-resistant actors Responsibility: 1. Decrements non-poison-resistant actors by a specified damage for a specified number of ticks 2. If actor is unconscious due to to external factors, the poison effect is removed 3. Similarly, if the actor dies due to the poison, poison effect is also removed

PoisonSwamp (extends Ground)	A ground the represents a poison swamp Responsibilities: 1. Checks if there's an actor standing on top of the Poison Swamp, adds a Poison Effect to any actors if so
EntityPassiveAbility	Added a new Enums: 1. POISON_RESISTANCE, which represents actors that are immune to the poison effect 2. GAMEOVERCAPABLE, which represents actors that can end the game.
MovePlayerAction (extends MoveActorAction)	Catches the exception when another Actor is standing on the destination
Player	Overrides the unconscious method to handle player's unconscious state when defeated by an actor

Solution and Explanation:

In Application, the maps Belurat Tower and Belurat Sewers are represented as arrays, before instantiating a new GameMap object using the groundFactory object. To instantiate the Belurat Sewers map, groundFactory has to be modified to include PoisonSwamp as well.

The creation of Gate is inspired by the Rocket and Window in the Mars demo code, where we add a MoveActorAction to move an actor into a different map. Gate is a Ground mainly for the characteristics shown in the expected output, where the Player is able to travel from the surroundings of the gate. First, we extend the Ground abstract class, and override the allowableActions method to add a MoveActorAction to allow travel between maps. The locations the gate is capable of transporting the Players to have to first be added by using the addLocations method which adds the location into the ArrayList attribute, locations in the class. The allowable actions method then iterates over the locations ArrayList and adds a MovePlayerAction accordingly. As a safety measure, there is a safeguard in MovePlayerAction to check if the destination

contains an actor, not allowing travel when there is one. Lastly, add EntityPassiveAbility.FIRE_RESISTANCE to not allow the gate to catch on fire.

PoisonEffect mainly works by leveraging the tick method from its parent class, StatusEffect. First, the constructor is modified to accept input, poisonDuration and damage. Then, the tick() method will first check if the actor has the POISON_RESISTANCE capability, removing the PoisonEffect if so. Subsequently, it checks if the age of the poison has reached the poisonDuration, removing the PoisonEffect from the actor when reached. If the poisonDuration has not been reached, it will decrement the actor's health by the specified damage, followed by incrementing the age and a check to see if the actor is unconscious. Lastly, we check if the actor is dead after the poison effect, removing the poison effect if they're dead.

PoisonSwamp is a ground that (currently) only exists in Belurat Sewers. For every tick, it will check if there are actors standing on top of the ground, applying a PoisonEffect onto the actor.

Advantages:

1. SOLID Principles

   Single Responsibility Principle (SRP)

- Gate specifically handles transporting actors into different maps
- PoisonEffect handles the application and the effect of the poison on actors, instead of having all the logic of PoisonEffect in PoisonSwamp
- PoisonSwamp only handles the logic of checking actors that are on top of the PoisonSwamp

   Open-Closed Principle (OCP):

- The Gate class can be extended to handle more locations in the future
- The PoisonEffect can be modified by adding more capabilities or poison behaviour

   Dependency Inversion Principle (DIP)

- Instead of directly depending on concrete classes, actions like applying poison or moving actors are decoupled and handled by general-purpose methods (allowableActions(), tick()), promoting flexibility.

2. Connascence

   The design demonstrates low coupling between components:

   - The grounds in the maps (Belurat Tower, Belurat Sewers) are created via the GroundFactory, which decouples the logic for creating different ground types like PoisonSwamp.
   - Adding new behaviours like the PoisonEffect doesn't require modifying existing actor classes; instead, poison is implemented through the tick() method, keeping connascence of execution low.
   - The MoveActorAction ensures safety through its safeguard mechanism, meaning the responsibility is bound within the method itself

3. DRY Principle
   - The PoisonEffect logic is encapsulated in the StatusEffect parent class, so the ticking mechanism is not repeated elsewhere.
   - The Gate class handles movement between maps in a reusable way, meaning other forms of inter-map travel can use the same mechanism with different configurations.
   - Any ground type that applies poison effects like PoisonSwamp can reuse the logic already built into the PoisonEffect mechanism
4. Extensibility
   - New locations or travel mechanisms can be added through the addLocations() method in the Gate class without having to rewrite any travel logic.

Possible Limitation/Drawback

1. Large Location Lists in Gate

- The allowableActions() method iterates over the list of locations to determine available travel options. As the list of destinations grows, this could lead to performance issues. When iIterating through a large list of locations for every player action could slow down the game, especially in scenarios where multiple players interact with the gate simultaneously.

2. Not future proofed for multiple poison types

- PoisonEffect currently implements a single type of poison, which makes it harder to introduce variations (e.g., stronger poison, faster-acting poison).

3. Hardcoded Behavior for Applying Poison

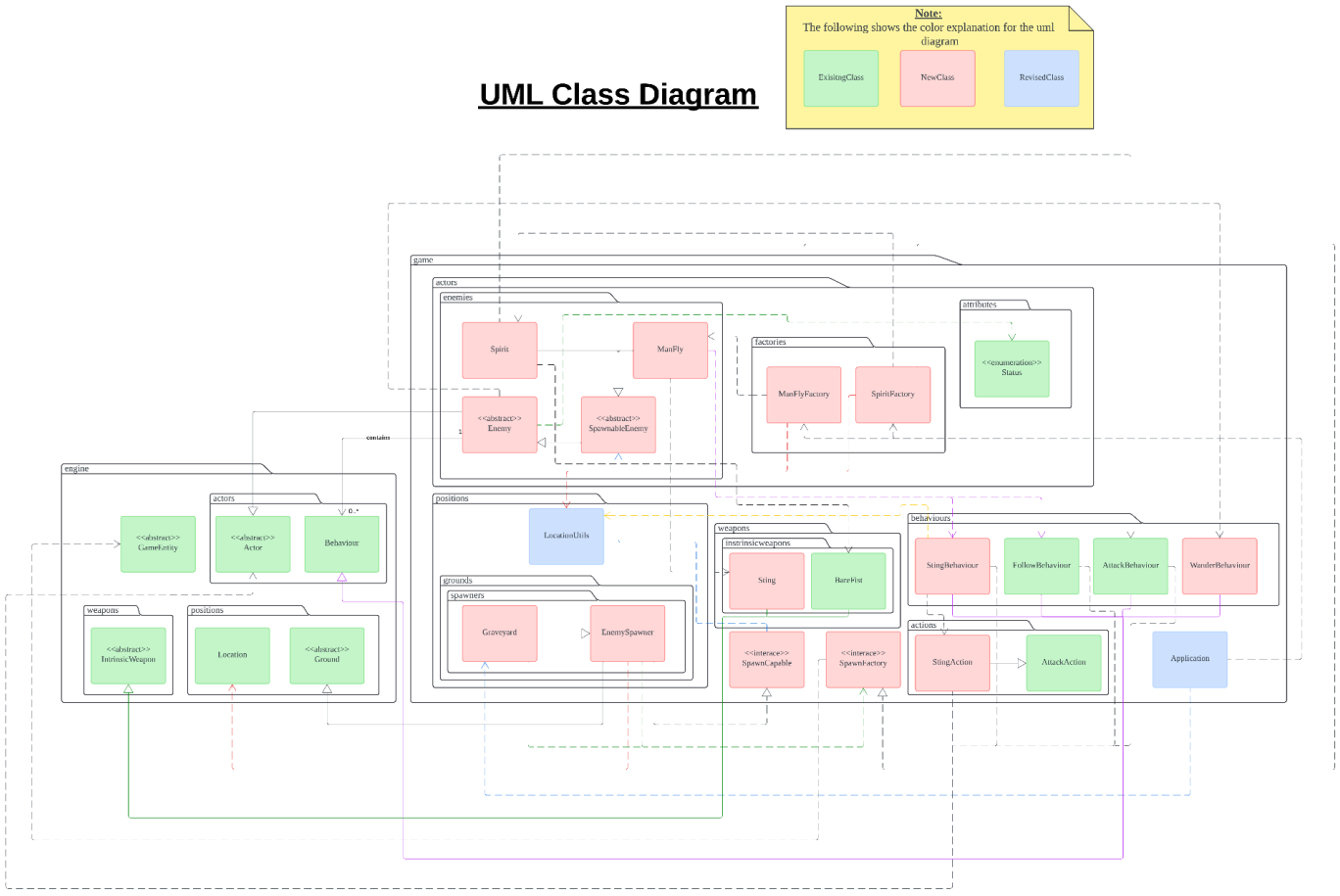
- The PoisonSwamp directly applies a specific PoisonEffect to actors standing on it, which makes it hard to reuse or modify this behavior for different types of effects.

Req 3: Ailment

UML Class Diagram

MONASH University

UML Class Diagram



Link to uml & sequence diagram for req3:
https://lucid.app/lucidchart/391f3bd6-0b50-4ad9-9a46-ce773b3e3ddb/edit?invitationId=inv_d5e3147b-2349-4d9f-84e2-614bfa99b9f9

MONASH University

Req 3 Design Rationale

Classes Created / Modified	Roles and responsibility
Application	Added new 3 grounds for each new map: i) Graveyard with Spirits to spawn ii) Graveyard with ManFly to spawn
ManFly (extends SpawnableEnemy)	Responsibility: 1. ManFly class represents the enemy man fly which stings and follows the player 2. It also has the ability to poison the player at 30% if stung Relationships: 1. Extends SpawnableEnemy since this is an enemy which could be spawned by a GameEntity 2. Has a Factory to create an instance of ManFly (ManFlyFactory) 3. Uses the PoisonEffect for poisoning the Player
Spirit (extends SpawnableEnemy)	Responsibility: 1. Spirit class represents the enemy spirit which attacks the Player with their bare fists Relationships: 1. Extends SpawnableEnemy since this is an enemy which could be spawned by a GameEntity 2. Has a Factory to create an instance of Spirit (SpiritFactory)
ManFlyFactory (implements SpawnFactory)	Responsibility: 1. ManFlyFactory class specialises in creating ManFly instances

	Relationships: 1. Implements SpawnFactory since SpawnFactory is in charge of creating GameEntity instances
SpiritFactory (implements SpawnFactory)	Responsibility: 1. SpiritFactory class specialises in creating Spirit instances Relationships: 1. Implements SpawnFactory since SpawnFactory is in charge of creating GameEntity instances
Sting (extends IntrinsicWeapon)	Responsibility: 1. Sting is an intrinsic weapon that deals 20 damage with 25% chance used by ManFly Relationships: 1. Extends IntrinsicWeapon since it is the default weapon of ManFly
StingAction (extends AttackAction)	Responsibility: 1. StingAction applies the poison effect for ManFly with its specific settings Relationships: 1. Extends AttackAction since we use both intrinsic weapon and poison to deal damage to actor
StingBehaviour (extends Behaviour)	Responsibility: 1. StingBehaviour represents the ManFly's attack of Sting which does a poison effect and also the sting damage caused by its

	intrinsic weapon with the same name Relationships: 1. Extends behaviour since it uses engine Behaviour methods
SpawnCapable (interface)	Responsibility: 1. An interface that represents any GameEntity that places a SpawnableEnemy at a specific Location Relationships: 1. It is an interface implemented by EnemySpawner
SpawnFactory (interface)	Responsibility: 1. An interface that represents a class (factory) that creates an instance of any GameEntity Relationships: 1. It is an interface implemented by ManFlyFactory and SpiritFactory
EnemySpawner (extends Ground & implements SpawnCapable)	Responsibility: 1. EnemySpawner is a ground that spawns a specified type of SpawnableEnemy at a given location based on a spawn chance Relationships: 1. It extends Ground since it requires tick and cannot ever be used like an item 2. Implements the interface SpawnCapable since it is a GameEntity that places a SpawnableEnemy at a specific Location

MONASH University

Graveyard (extends EnemySpawner)	Responsibility: 1. Spawns Spirit at Belarut Tower and ManFly at Belarut Sewers Relationships: 1. It extends EnemySpawner since it spawns SpawnableEnemy but it has different attributes to EnemySpawner (name, display character, etc)
SpawnableEnemy (abstract & extends Enemy)	Responsibility: 1. SpawnableEnemy is an abstraction of Enemy that contains extra attributes for Enemy class that can be spawned Relationships: 1. It is an abstract class since it will not be instantiated 2. Extends Enemy class since the actors are still enemies
Enemy (abstract & extends Actor)	Responsibility: 1. Enemy class is an abstraction of Actor class that handles enemy like behaviour like WanderBehaviour and status of being friendly towards each other Relationships: 1. It is an abstract class that all actors opposing Player should extend from 2. Extends from Actor since it is still an NPC and we require its methods and attributes
WanderBehaviour (extends Behaviour)	Responsibility: 1. In charge of NPCs being able to move around randomly in the map

MONASH University

	Relationships: 1. Extends Behaviour class like all other Behaviours

Solution and Explanation:

Problem:

Create a graveyard Ground that spawns ManFly & Spirit at Belurat Sewers and Belurat Tower respectively with their respective behaviours.

Solution:

1. Create a Ground especially for spawning enemies (EnemySpawner), Graveyard class and SpawnCapable interface to ensure SOLID principles to solve Graveyard
2. Spirit & Manfly classes are solved by creating an abstraction of Enemy class (SpawnableEnemy), SpawnFactory interface and factory classes (ManFlyFactory & SpiritFactory), Spirit and ManFly classes to solve spawning these 2 enemies
3. StingAction, StingBehaviour and Sting is created for ManFly's requirements for its stipulated requirements including poison

Explanation:

1. We have to create a class of Ground type that can spawn any GameEntity
    a. First we create an interface called SpawnCapable that must be implemented by a class that spawns any SpawnableEnemy with the method *void spawn(Location location, SpawnableEnemy enemy)* given a Location & SpawnableEnemy argument
    b. Next create an abstraction of Ground that uses the previous step's interface called EnemySpawner which has a modified *tick* method that spawns an enemy at a location of a SpawnableEnemy. EnemySpawner takes in a class that implements SpawnFactory as an argument and that is how we also obtain the SpawnableEnemy that is to be spawned
    c. Finally, we create a Graveyard class which is a subclass of the previous step's EnemySpawner that is able to spawn any EnemySpawner at its current location (where the Graveyard is located) given a class that implements SpawnFactory

2. To spawn ManFly & Spirit enemies from another GameEntity unlike FurnaceGolem from A1
    a. First we create an abstraction of Actor class (Enemy) for all NPC opposing Player and another abstraction of the Enemy class called SpawnableEnemy so that we can store attributes and methods for Enemies that are able to spawn like it's *SpawnChance*
    b. Next we create ManFly and Spirit class extended from SpawnableEnemy (from the previous step) with their stipulated attributes and information from the requirements
    c. Then to actually spawn the created classes from the previous step we have to create a SpawnFactory interface which has method *createGameEntity()* that creates an instance of any GameEntity. After that, create ManFlyFactory & SpiritFactory classes that implement SpawnFactory in order to create an instance of ManFly and Spirit

3. In order to make ManFly be able to sting and poison another actor we have to use existing implementations from previous requirements
    a. First we create the following classes for stinging; StingAction that extends AttackAction and adds a new PoisonEffect to the target Actor, StingBehaviour extends Behaviour and returns a StingAction to perform stinging and finally creating Sting intrinsic weapon class which just handles the sting basic damage and chance of hit
    b. PoisonEffect is integrated with stinging by using *target.addStatusEffect(new PoisonEffect(POISON_DURATION, POISON_DAMAGE))* which uses an implementation from the previous requirement
    c. Note that Player and other NPCs have different poison decrement operations. That is, Player health decrements all at once (ie: PoisonEffect + Attack from Enemy is printed in one line) whereas ManFly handles this logic slightly differently by showing each instance of where it's health is decremented (ie: Prints HP deduction from PoisonEffect, then a separate line for an attack from Player)

Advantages:

1. SOLID Principles

    Single Responsibility Principle (SRP)

- ManFly & Spirit classes handles their own behaviours with their own methods and attributes
- ManFlyFactory & SpiritFactory also handles their own creation of their respective SpawnableEnemy instances independently
- StingAction, StingBehaviour & Sting intrinsic weapon class all also has a single responsibility to fulfil stinging like in A1

Open-Closed Principle (OCP)

- Without modifying any existing classes we are able to add more enemies that are able to be spawned by extending SpawnableEnemy which means open to extending to more spawnable enemies and close to modifying existing classes everytime we extend

Liskov Substitution Principle (LSP)

- A very good example of LSP used is EnemySpawner which is used in req4 to spawn Scarab class but is also generally used for extending classes like Graveyard which spawns ManFly & Spirit classes in req3. This parent (EnemySpawner) and its subclass (Graveyard) can be used interchangeably whilst make sure that they perform the same function the same way thus satisfying LSP
- ManFly & Spirit classes are also interchangeable with its parent class of SpawnableEnemy thus adhering to LSP

Interface Segregation Principle (ISP)

- Having SpawnCapable interface allows for focusing on one job which is to create instances of SpawnableEnemy without any other unnecessary methods

Dependency Inversion Principle (DIP)

- High level modules do not depend on low level modules; for our implementation high level modules like SpawnableEnemy do not depend on low level modules like ManFly or Spirit

2. Connascence

The implementation for req3 aims for low Connascence and does so in the following:

- Connascence of Name (CoN): ManFlyFactory must create ManFly objects, their names are tightly coupled with the objects they create (Strength: weak)
- Connascence of Type (CoT): EnemySpawner's arguments only accepts SpawnableEnemy entity (Strength: moderate)
- Connascence of Meaning (CoM): Values like SpawnChance if changed cannot be interpreted the same way (Strength: moderate)

- Connascence of Algorithm (CoA): Don't really have major CoA since changing most algorithms won't cause the whole system to have major errors
- Connascence of Execution (CoE): Since *tick* is completely handled by the engine and is out of scope CoE is avoided. But for poisoning and initial damage by the ManFly we need to adhere to a certain order which it does by executing the AttackAction first then PoisonEffect

Hence, we observe that this implementation indeed has low coupling.

3. DRY Principle
    - SpawnableEnemy prevents code duplication for attributes and methods for enemies that can be spawned as an abstract class

4. Extensibility (Answering Design Considerations in req3)
    - There could be graveyards in other maps spawning other entities
        a. Since Graveyard class is implemented as extending EnemySpawner it is able to spawn any type of enemies given a factory class of said enemy. It can also be placed anywhere in the map in Application like any other Ground object
        b. Note: Graveyard is not fire resistant so it will turn into a fire and not spawn (future implementation could make Graveyard or EnemySpawner fire resistance or combine both effects in another implementation)

    - There could be new spawners in Belurat and its sewers (and other maps), spawning new entities
        a. EnemySpawner class can be extended to create new spawners besides Graveyard as implemented SpawnCapable interface ensures that any extended class from EnemySpawner must have a method to spawn SpawnableEnemy entities. Again like any other Ground (since EnemySpawner extends from Ground) the newly created spawners can be placed anywhere in any map

- In the future, there is a possibility that an actor can spawn another actor, so "spawning" is not an ability limited to grounds.
    a. With the combination of SpawnFactory (creates an instance of GameEntity) and SpawnCapable (spawns entities) we are able to solve this problem as we can just implement the abstraction to another newly created class e.g. ActorEnemySpawner which is an actor that spawns SpawnableEnemies and its implementation to spawn entities will be the same as existing class EnemySpawner

Possible Limitation/Drawback

- For the current requirements for A2 the factory implementation introduces additional code complexity and can actually be solved using less abstraction, classes and methods (Drawback)
- The current factory implementation has a huge overhead where whenever we add a new SpawnableEnemy we have to create a new factory class for it which violates the OCP principle and DRY principles (Drawback)
- Limitation for a very SRP class of StingBehaviour has a limitation for its reusability which can lead to potential code duplication (Limitation)
- ManFly is tightly coupled with its StingAction & StingBehaviour which means if we want to edit its behaviour tremendously we have to change the class implementation drastically (limitation)

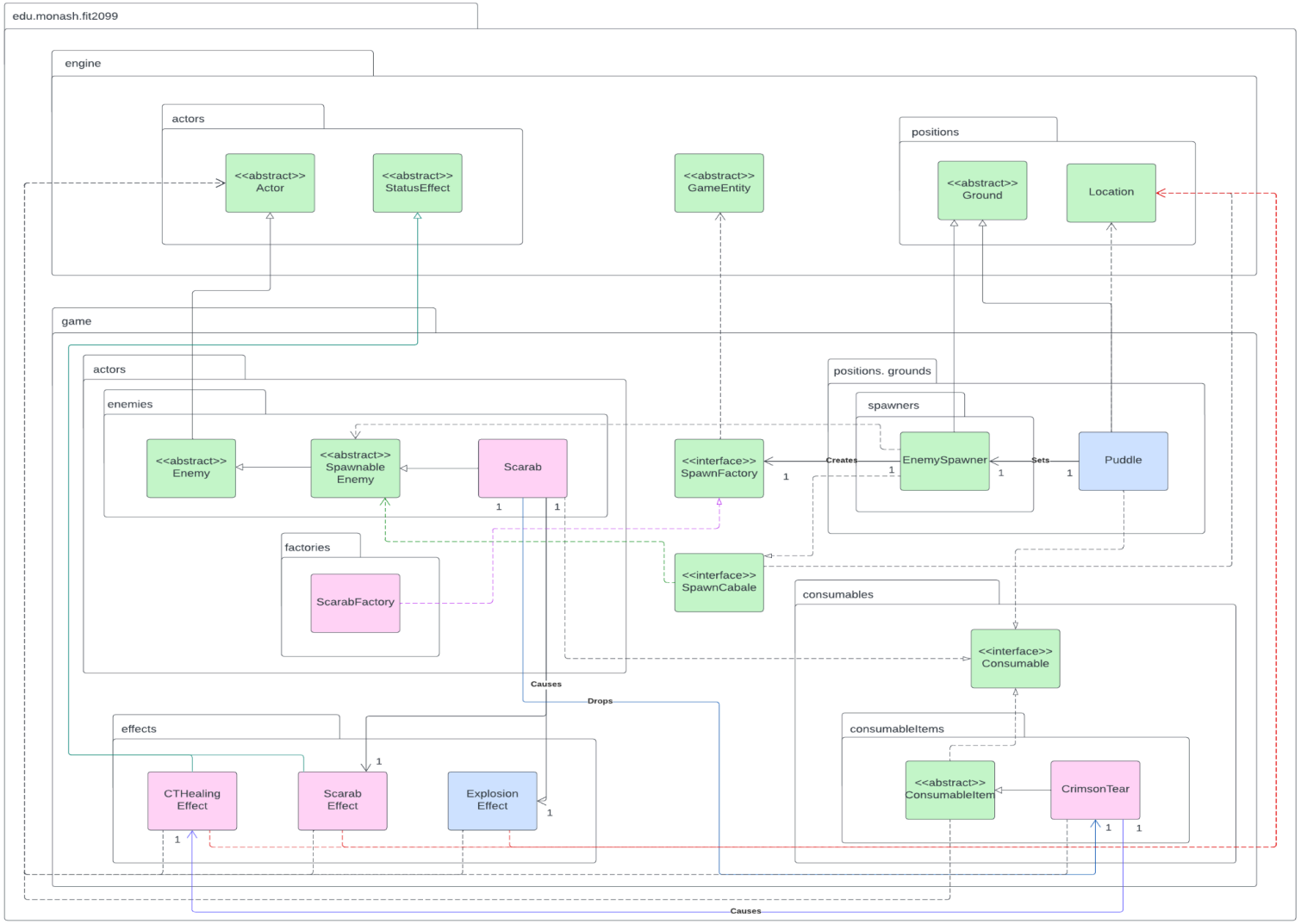
Alternative Solution

- Instead of using the factory implementation we could directly create instances of the SpawnableEnemy inside the class itself which reduces code complexity as well as dependencies

MONASH University

Req 4: Be wary of dung (beetle)

Req 4 UML Class Diagram



MONASH University

Req 4 Design Rationale

Classes Created / Modified	Roles and responsibility
Scarab (extends SpawnableEnemy) (implements Consumable)	A class that represents a Scarab SpawnableEnemy. Responsibilities: 1. Extends SpawnableEnemy abstract class as it possesses a wandering behaviour as well as interact with other actors (can be attacked by Player) - creates an ExplosionEffect when it is unconscious. 2. Implements Consumable interface as it can be consumed by Player when it is near the Player's surroundings - creates a ScarabEffect buff to the Player.
CrimsonTear (extends ConsumableItem)	A class that represents a CrimsonTear consumable item. Responsibilities: 1. Extends ConsumableItem abstract class to provide a concrete example of a consumable item that can be picked up and consumed. 2. Creates a CTHealingEffect to actors that consume this consumable item.
ScarabEffect (extends StatusEffect)	A class that represents a Scarab Effect to the actor that consumes a Scarab. Responsibilities: 1. Extends StatusEffect abstract class to provide a concrete example of a status effect that can be applied to actors that consume a Scarab 2. The actor's maximum health and mana will be increased by 30 and 50 respectively for 10 turns

MONASH University

CTHealingEffect (extends StatusEffect)	A class that represents a Healing Effect to the actor that consumes a Crimson Tear consumable item. Responsibilities: 1. Extends StatusEffect abstract class to provide a concrete example of a status effect that can be applied to actors that consume a CrimsonTear item. 2. Heals the actor for 30hp for 5 turns
ExplosionEffect (implements DamageEffect)	A class that represents an Explosion Effect to the surroundings of a location. Modifications: 1. Removed final int of damage and modified the constructor to accept a parameter (damage) as previously only StompAction causes this effect hence the damage was final. However, Scarab can cause this effect but with a different amount of damage.
Puddle (extends Ground) (implements Consumable)	A class that represents a type of ground object in the map. Modifications: 1. Implements Consumable now as Actors can drink water when standing on top of a puddle ground. 2. In every tick now, there is a condition to check if the puddle is consumed and if a Scarab spawns on the surroundings of the puddle that was consumed
ScarabFactory (implements SpawnFactory)	A class that represents a type of SpawnFactory to create instances of the Scarab enemy Responsibilities: 1. Create Scarab enemy and injecting an explosive effect in the Scarab constructor

MONASH University

Solution and Explanation:

The implementation of Scarab is similar to the previous enemies however the difference is Player has the option to consume Scarab instead of attacking it. Hence, to make this possible, Scarab class is made to implement the Consumable interface to obtain the consume method as well as having a consumeAction whenever a Player is near the scarab. Two effects known as Scarab Effect which can be obtained by consuming the Scarab and CTHealing effect which can be obtained by defeating the scarab and consuming the CrimsonTear item that is dropped. These two effects are created as subclasses of StatusEffect to be able to access the tick() method. This is crucial as the buff effects that are given to the Player apply through multiple playturns. This cannot be achieved in the Scarab playturn method itself unless it is done by adding a statusEffect to the Player. There is a tick method to keep track of the number of ticks passed in both the effect classes so that the StatusEffect can be removed from the Player once it finishes the duration of turns.

Another class that is modified to implement the Consumable interface is the Puddle class. Now, the task given is to give the option to the Player to consume the puddle where the Player is standing on, and not the surrounding puddles. This is easily achieved by overriding the allowableActions method where the location checks if there is an actor standing on the puddle (possible to check as puddle is a subclass of Ground abstract class), if the condition is true, consumeAction is added to the possible actions list in the method. The method: consume() can be accessed due to implementing the Consumable interface.

A big modification in Puddle is done in the tick method(). To be able to spawn a Scarab, a possible location for it to spawn must be checked. By getting all possible exits from the puddle that it consumed, one puddle is picked and checks if the chances of the Scarab being spawned is fulfilled. After these two conditions are true, the puddle that is consumed will then be set to spawn a ScarabFactory (EnemySpanner class will be in charge to spawn a ScarabFactory). Hence, the Scarab is created by using the factory's createGameEntitiy method. Once the Scarab is spawned, the ground returns to puddle.

MONASH University

Advantages:

Open/Closed Principle (OCP)

The StatusEffect class allows extensions via subclasses like ScarabEffect and CTHealingEffect. These subclasses add new functionality for tracking turns and applying effects without modifying the original StatusEffect class. This adheres to the OCP since new functionalities can be added by subclassing instead of modifying the core class.

Liskov Substitution Principle (LSP)

The Consumable interface is a good example of LSP in action. Both the Scarab and Puddle classes can be substituted for any other class implementing the Consumable interface. A Player interacting with either a Puddle or Scarab can safely call consume() without needing to know the specific implementation.

Interface Segregation Principle (ISP)

By converting the SpawnFactory from an abstract class into an interface, the EnemySpawner class has the flexibility to depend only on the methods of the interface that it actually needs to create actors, without inheriting unnecessary functionality from a larger, multi-purpose abstract class. This adheres to ISP and allows ScarabFactory to access the createGameEntity method as the class implements SpawnFactory.

Dependency Inversion Principle (DIP)

ScarabFactory and EnemySpanner classes handle the instantiation and spawning of enemies like Scarab. By using a factory pattern, the creation of the Scarab is abstracted, and higher-level modules (such as the puddle that spawns the Scarab) are not directly dependent on the instantiation details of the Scarab. Instead, they depend on an abstraction (i.e., the factory's createGameEntity() method).

By defining interactions like consume() through the Consumable interface, higher-level classes (like Player) depend on abstractions rather than concrete implementations of Scarab or Puddle. This approach adheres to the DIP by ensuring that details (concrete classes) depend on abstractions (interfaces), not the other way around.

MONASH University

Connascence

The degree of coupling is low to moderate because:

- The use of the Consumable interface and StatusEffect subclasses introduces a type dependency. However, these types are well-abstracted through interfaces and inheritance, which makes it easier to modify or extend the system without breaking functionality. The design supports flexibility and extensibility.
- Similarly to REQ2, Adding new StatusEffect like the ScarabEffect and CTHealing doesn't require modifying existing actor classes; they are implemented through the tick() method, hence keeping connascence of execution low.

Possible Limitation/Drawback

The current factory implementation has a huge overhead where whenever we add a new SpawnableEnemy we have to create a new factory class for it which violates the OCP principle and DRY principles (Drawback).

Similarly to REQ3, the drawback for this requirement is how ScarabFactory class has to be made first in order to create an instance of a Scarab when using factory pattern implementation.

Hence, instead of using the factory implementation we could directly create instances of the SpawnableEnemy inside the class itself which reduces code complexity as well as dependencies (Same solution as REQ3).