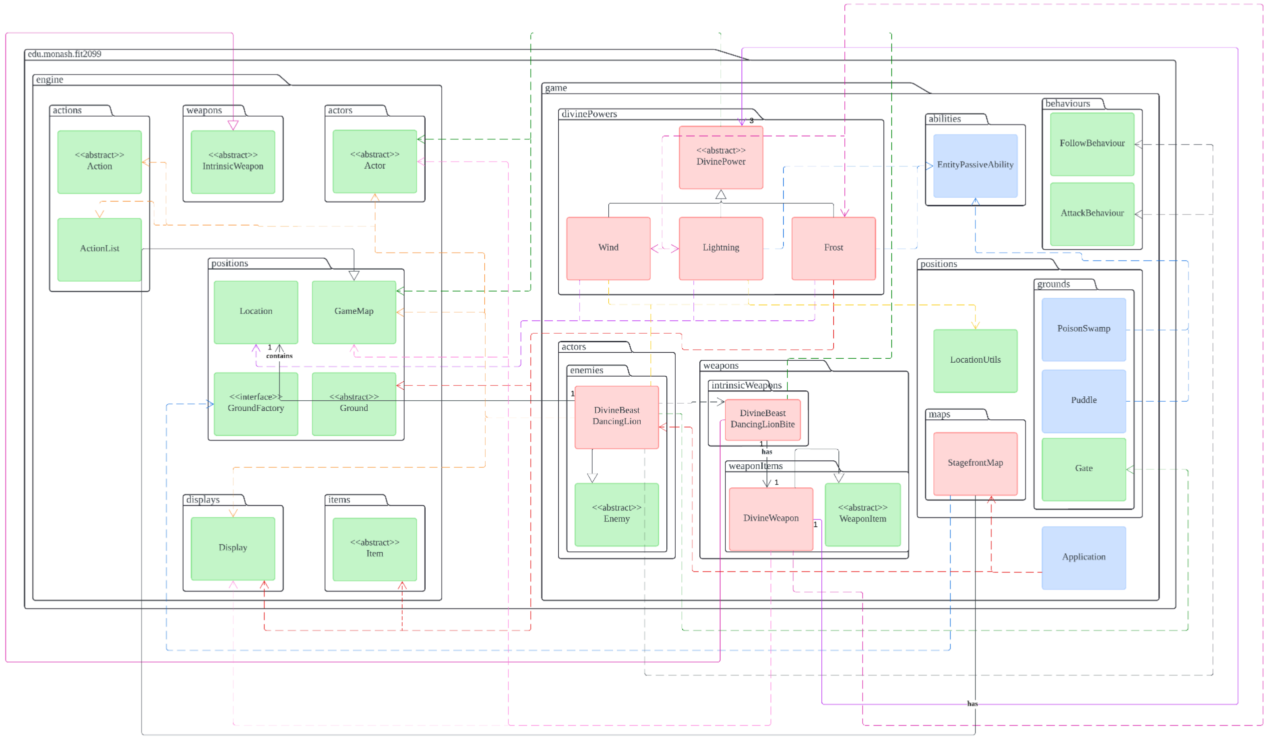
Assignment 3 Design Rationale

REQ 1 UML Class Diagram

UML Class Diagram

Note:
The following shows the color explanation for the uml diagram

🟩 ExistingClass 🟥 NewClass 🟦 RevisedClass



Link to uml & sequence diagram for req1:
https://lucid.app/lucidchart/d47ff290-5fb0-4a5f-a94e-875b571efb27/edit?viewport_loc=-979%2C-2978%2C9307%2C4290%2C0_0&invitationId=inv_b2e0ad9f-9cbf-4a5b-9c3c-625ec9b1b5c7

Req 1 Design Rationale

Classes Created / Modified	**Roles and responsibility**
Application - modified	Modified the following: 1. Added a new Stagefront map introduced in A3 requirement 1 to Application 2. Added graveSiteToStagefrontGateLocation (type Location) which represents the Location which the Gate from Gravesite to Stagefront will be located 3. Added a new instance of DivineBeastDancingLion which takes in the graveSiteToStagefrontGateLocation created earlier 4. Added stagefrontArriveLocation (type Location) which represents the Location which the Player arrives at in the map Stagefront from Graveyard 5. Added a new Gate in Graveyard map called graveSiteToStagefrontGate which sends the Player from graveSiteToStagefrontGateLocation in Graveyard map to stagefrontArriveLocation in Stagefront map
DivinePower (abstract) - created	Responsibility: 1. DivinePower abstract class represents the blueprint for which all the divine powers should extend from 2. Contains methods and attributes for the DivinePower's information (i.e. name) as well as an method *useDivinePower* that all DivinePower must have

	Relationships: 1. An abstract class which is extended by the 3 divine powers (i.e. Wind, Lightning & Frost)
Frost (extends DivinePower) - created	Responsibility: 1. Frost class represents the Frost divine power described in the requirement which makes the Player drop all the items in their inventory if standing on a Ground with EntityPassiveAbility's *PUDDLE_OF_WATER* Relationships: 1. Extends DivinePower since it is a divine power that can be used by a GameEntity
Wind (extends DivinePower) - created	Responsibility: 1. Wind class represents the Wind divine power described in the requirement which makes the Player move to another spot adjacent to the GameEntity using the DivinePower Relationships: 1. Extends DivinePower since it is a divine power that can be used by a GameEntity
Lightning (extends DivinePower) - created	Responsibility: 1. Lightning class represents the Lightning divine power described in the requirement which makes the Player or any Actors adjacent of the GameEntity using the DivinePower take 50 damage or 100 damage if the Player or any other adjacent Actors are standing on a Ground with EntityPassiveAbility's *PUDDLE_OF_WATER* Relationships: 1. Extends DivinePower since it is a divine power that can be used by a GameEntity
DivineBeastDancingLion (extends Enemy) - created	Responsibility: 1. DivineBeastDancingLion represents the Enemy boss in the Stagefront map that is

	capable of using DivinePower 2. When unconscious DivineBeastDancingLion will also turn into a Gate to return to its set Location it takes in as an argument 3. FollowBehaviour also added to its list of behaviours as it is able to follow the Player Relationships: 1. Extends Enemy as it is classified as an attackable NPC 2. Has dependency with previous assignment requirements like Gate or FollowBehaviour
DivineBeastDancingLionBite (extends IntrinsicWeapon) - created	Responsibility: 1. This newly created class represents the IntrinsicWeapon used by the DivineBeastDancingLion essentially representing a “bite” Relationships: 1. Extends IntrinsicWeapon since it is a default weapon used by DivineBeastDancingLion to deal damage to other GameEntites 2. It also instantiates a new DivineWeapon by passing in the static final attributes from DivineBeastDancingLion as arguments allowing it to use Divine Power
DivineWeapon (extends WeaponItem) - created	Responsibility: 1. DivineWeapon class allows Actors to use DivinePowers as well as handles switching logic Relationships: 1. Extends WeaponItem since it could be dropped and picked up
StagefrontMap (extends GameMap) - created	Responsibility: 1. StagefrontMap represents the new map that is added in REQ1 in A3 called Stagefront

	Relationships: 1. Extends GameMap since it has all the capabilities of a GameMap that needs to be performed
Puddle (extends Ground implements Consumable) - modified	Modified the following: 1. Added new capability of EntityPassiveAbility's PUDDLE_OF_WATER
PoisonSwamp (extends Ground) - modified	Modified the following: 1. Added new capability of EntityPassiveAbility's PUDDLE_OF_WATER
EntityPassiveAbility (enumeration) - modified	Modified the following: 1. Added new entry of PUDDLE_OF_WATER to represent a characteristic of a puddle of water

<u>Solution and Explanation:</u>

Problem:

1. Need to create & add new map Stagefront to existing world and connect it to existing map Gravesite
2. Need to introduce new concept of DivinePower to the game mechanics and add the stipulated new divine powers (Frost, Wind & Lightning)
3. Need to add a way for the Actor to use DivinePower, DivineWeapon
4. Need to create & add new Enemy that can use DivinePower, DivineBeastDancingLion

Solution:

1. Create and add a new map to the existing world called **Stagefront** with the Gate from Gravesite to Stagefront in **Application**
2. Create new **DivinePower** being **Frost**, **Wind** & **Lightning** with modification to **Puddle**, **PoisonSwamp** & **EntityPassiveAbility**
3. Create **DivineWeapon** where it handles all the DivinePower usage and switching logic and any actor using it can use divine powers
4. Create and add new Enemy **DivineBeastDancingLion** with its Gate requirements and creating its IntrinsicWeapon of **DivineBeastDancingLionBite** where it contains an instance of DivineWeapon allowing the Lion to use divine powers

Explanation:

1. Create and add a new map to the existing world called **Stagefront** with the Gate from Gravesite to Stagefront in **Application**
    a. First we create StagefrontMap class to represent the new map of Stagefront that is to be added into the World
    b. Second we add the Array of symbols provided in the requirement to the Application script
    c. Finally we create a new instance of StagefrontMap using the array of symbols hence adding a new map to the world

2. Create new **DivinePower** being **Frost**, **Wind** & **Lightning** with modification to **Puddle**, **PoisonSwamp** & **EntityPassiveAbility**
    a. First in order to create the different required divine powers (Frost, Wind & Lightning) we need to create an abstraction for the divine powers to extend from, hence DivinePower abstract class is created
    b. Then we proceed to create Wind divine power first, where we implement the Wind logic by overriding the method *useDivinePower* from the DivinePower abstract class. Its logic is handled by getting the accessible Locations of the DivinePower user and moves the target Actor to a random accessible Location using moveActor method from GameMap
    c. Before proceeding with implementing Frost and Lightning logic we first add a new entry *PUDDLE_OF_WATER* to the enum EntityPassiveAbility and add capability of the new entry to Puddle and PoisonSwamp
    d. Next we proceed to create Frost divine power, where we implement the Frost logic by overriding the method *useDivinePower* from the DivinePower abstract class. Its logic is handled by checking if the target Actor's Ground has capability PUDDLE_OF_WATER, if it has then each item of its inventory will be removed and added to the current ground its located on (items will be stacked in the same Location)
    e. Finally we create Lightning divine power, where we implement the Lightning logic by overriding the method *useDivinePower* from the DivinePower abstract class. Its logic is handled by checking the surrounding (adjacent) locations of the divine power user and checks if there is an Actor or not, if it has then deal 50 damage to said Actor and if the target Actor's Ground has capability PUDDLE_OF_WATER then deal 100 damage instead

3. Create **DivineWeapon** where it handles all the DivinePower usage and switching logic and any actor using it can use divine powers
    a. The decision to make DivineWeapon a WeaponItem is to accommodate for REQ2 where the DivinePower can be "transferred" from the lion to the Player

b. First the DivineWeapon has an overridden method from WeaponItem which is *attack(Actor attacker, Actor target, GameMap map)*. This method handles the special attack aspect of the divine power requirement where it first has 25% chance to switch into another DivinePower using *switchDivinePower()* method then proceeds to invoke *useDivinePower(attacker, target, map)* from the chosen DivinePower after all the switching. Then finally proceeds to use its super's *attack* method
c. Second DivineWeapon's *switchDivinePower* method works using a cumulative probability approach since the probability given for switching is a distributed probability that adds up to 100. So it works by choosing a random number from 0 to 99 which represents a chosen probability and the cumulative probability will start from 0 and keep adding the probabilities from the transition (e.g. 0+60% = 60% [WIND] → 60%+40%=100% [LIGHTNING]). If the selected probability is less than cumulative probability then we switch to that DivinePower. Which means if our selected probability is 90% we selected LIGHTNING to switch to using the example given before since 90 is not less than 60 but is less than 100
d. Third we use the method *setupDivinePowerTransitions()* in order to hard code the possible transition between DivinePower and their probabilities. It also uses *addDivinePowerTransition(DivinePower currentPower, DivinePower nextPower, int chance)* to add to a mapping of *Map<DivinePower, Map<DivinePower, Integer>>* to keep track of all the possible transitions

4. Create and add new Enemy **DivineBeastDancingLion** with its Gate requirements and creating its IntrinsicWeapon of **DivineBeastDancingLionBite** where it contains an instance of DivineWeapon allowing the Lion to use divine powers
    a. DivineBeastDancingLion is created from abstract class Enemy with its stipulated attributes
    b. DivineBeastDancingLion handles Gate requirement by overriding the unconscious method where a *stagefrontGate* is created in the current Location connecting to the Location DivineBeastDancingLion takes in
    c. DivineBeastDancingLionBite is created the intrinsic weapon for DivineBeastDancingLion in order for it to deal damage to other GameEntity and it also creates an instance of the DivineWeapon object in order for DivineBeastDancingLion to utilise DivinePower

<u>Advantages:</u>

1. SOLID Principles

    <u>Single Responsibility Principle (SRP)</u>

- SRP applies to each of the DivinePower (i.e. Wind, Lightning & Frost) since each DivinePower class focuses on their own operations like dropping items, moving Actors or dealing AOE damage

- DivineBeastDancingLion also follows SRP since it only handles DivineBeastDancingLion behaviours and information instead of handling the DivinePower operations inside it

Open-Closed Principle (OCP)

- Without modifying the existing abstract class of DivinePower we are able to extend divine power operations like adding new divine powers or modifying existing divine powers thus it follows OCP

Liskov Substitution Principle (LSP)

- DivinePower abstract class does adhere to LSP to a certain extent since it contains all the methods and attributes Wind, Frost & Lightning also has and hence can be used almost exchangeably if DivinePower weren't made an abstract class but concrete

- The same goes for DivineWeapon class where it can be used interchangeably with DivineBeastDancingLionBite since it passes in the necessary arguments and both are of weapon type

Interface Segregation Principle (ISP)

- No violation of ISP since no interfaces used for this requirement

Dependency Inversion Principle (DIP)

- Divine powers like Frost, Wind & Lightning depends on DivinePower which shows that high level modules depends on an abstraction instead of a low level module

2. Connascence

The implementation for REQ1 aims for low Connascence and does so in the following:

- Connascence of Name (CoN): Methods like useDivinePower relies on its method name which means, these methods are also used across several files across the system (i.e. Wind, Frost & Lightning) and the slight change of names will cause the system to fail (Strength: weak)

- Connascence of Type (CoT): Don't really have major CoT but if there were to be a variant of DivinePower like SuperDivinePower for only certain DivinePowers the methods like useDivinePower will not work anymore (Strength: moderate)

- Connascence of Meaning (CoM): The probabilities given for this requirement are all distributed probabilities which means if the probabilities are not longer distributed the current method cumulative probability would not work (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, except for the switching logic of DivinePowers using cumulative probability as a slight change in the methodology the whole transition logic will fail (strength: strong)

- Connascence of Execution (CoE): Don't really have major CoE but if the execution of special attack where the divine power is used after the basic attack instead we would have to refactor the implementation over again (strength: strong)

Hence, we observe that this implementation indeed has low coupling since only CoN and CoM are the only ones that are impactful

3. DRY Principle
    - DivineWeapon saves plenty of code repetition as it handles not just the execution of "special attack" but also transitions as well. All Actors have to do is just pick up the weapon and use it or create an instance of DivineWeapon with all the necessary arguments

    - DivinePower abstract class also saves a little code repetition as of now since we have little shared common attributes/methods for all divine powers

4. Extensibility (Answering Design Considerations in req3)

- **There may be other types of divine powers in the future.**
  a. We have the abstract class DivinePower that could be used as a blueprint to create more divine powers if needed. The new divine power logic will be handled in their own class

- **The dancing lion may not always start with the "Wind" divine power.**
  a. Currently not possible with the current implementation where the default is manually set in the class of DivineWeapon but is possible to implement it in the alternative method section below using the existing framework

- **(Challenge) the "Frost" divine power may link to "Lightning" in the future instead of "Wind". It may even link to some new divine powers, e.g., "Fire". Similarly, "Lightning" may not link to "Frost" or "Wind" if used by another instance of the dancing lion or other entities.**
  a. The switching of divine powers is handled using the method divinePowerTransition in class DivineWeapon where a sequence of switches between divine powers are added one by one and can be overridden by new Enemies to change the sequence of transitions

5. Code smell

- Brain class: No class is currently considered as a brain class but DivineWeapon is in charge of divine power execution and switching hence if either functionality increases it might grow into a Brain class

- Brain method: No class is currently considered as a brain method but if switching or divine power gets more complex methods like switchDivinePower might become a brain method

- Data Clumps: Data clumping might happen if we inject a fair amount of data into DivineBeastDancingLion class in order to fulfil the specification of having being able to set the initial divine powers

- Feature Envy: No feature envy smell but if DivineWeapon class has too many methods focusing on DivinePower instead of Weapon related functionality there might be feature envy smell

- God Class: Although currently not a God class, Application is more and more like a God Class where it handles creating a variety instances and other implementations

- Intensive Coupling: DivinePower and DivineWeapon classes are tightly coupled as a change in the logic in either class would require a change in the other. For example if the logic for applying the divine power were to change then DivineWeapon's switching or attack method would have to change

- Long method: The current methods for REQ1 implementation aren't too complex but if the switching between divine powers method gets too complex it is possible to have long method smell

- Shotgun Surgery: If the divine power logic changes drastically then the newly added classes like DivineWeapon will have to change its methods extensively

- Type Checking: No type checking code smell but if we have many different variants of DivinePower we might have Type checking code smell in the switching method

Hence, we observe that this implementation indeed has little to no code smell as the only serious smell would be Intensive Coupling

<u>Possible Limitation/Drawback</u>

- The current implementation for divine powers will make it so that the newly created classes are tightly coupled and will cause issues to the system if there are any major changes to the logic of divine powers and how game entities uses it since we need to add the newly created DivinePower switching probabilities as well as create a new instance of it in DivneWeapon class (limitation)

- If different switch logic needs to be implemented the current implementation will not work since it is a rigid non flexible map between one divine power to another (limitation)

- One of the most important limitations is that the instances of DivineBeastDancingLion must have the same sets of switching sequence/logic which means they must implement the switching of DivinePower the same way (limitation)

- Whenever we need to add a new switch between different divine powers we have to add it manually into the method which is a huge drawback (drawback)

- The current implementation of how we add new items/GameEntities to the world/map in Application only exponentially increases its coupling and dependencies (drawback)

Alternative Solution

- The most important thing to fix for the first alternative implementation is to make sure that different instances of the same class (e.g. DivineBeastDancingLion) are able to have different sets of switching between different divine powers and also be able to have different starting divine powers.

    a. This can be achieved by having an Enum with the entries corresponding to the names of the DivinePowers in the game and also instantiate the corresponding DivinePowers

    b. Then we do dependency injection from the Application class of the starting divine power as well as a list of mapping between the divine power and the next divine power including the probability of the switch using the Enum

    c. This also increases the dependencies of Application further

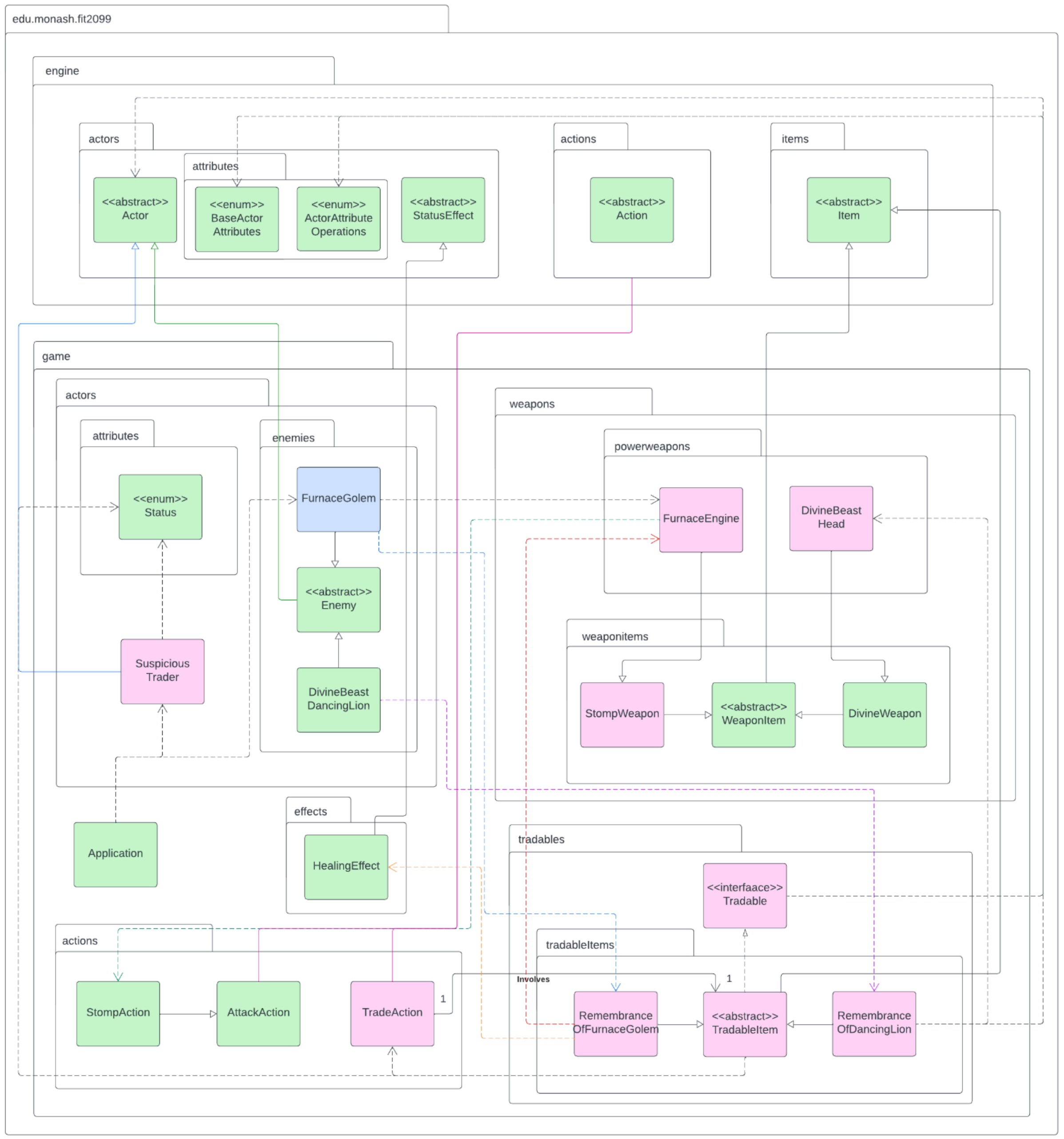
MONASH University

Req 2: Remembrance of a Legend

Req 2 UML Class Diagram

The following show the colour explanation for the UML diagram

🟩 ExistingClass 🟪 NewClass 🟦 ModifiedClass



MONASH University

<u>Req 2 Design Rationale</u>

Classes Created / Modified	**Roles and responsibility**
Application	Modification 1. Added instance of SuspiciousTrader at the start of the game.
SuspciousTrader (extends Actor)	A class that represents a SuspiciousTrader actor. Responsibilities: 1. Extends Actor abstract class as it interacts with other actors (can trade items with Player).
Tradable	An interface to be implemented by any objects that can be traded with SuspiciousTrader Responsibilities: 1. Allow any classes that implement this class to override trade method 2. Prevents allowing only TradableItem objects to be traded
TradableItem (extends Item) (implements Tradable)	An abstract class that represents a TradableItem Item Responsibilities: 1. Extends Item abstract class to provide a concrete example of an Item that has portability. 2. Implements Tradable interface class to be able to be traded by Player with SuspiciousTrader. 3. Able to override the trade method implemented from Tradable which can give Player benefits after trading a tradable item. 4. Allow TradeAction for all TradableItem objects.

RemembranceOfFurnaceGolem (extends TradableItem)	A class that represents a RemembranceOfFurnaceGolem TradableItem Responsibilities: 1. Extends TradableItem abstract class to provide a concrete example of a TradableItem. 2. Grants whoever has this item the capabilities of a FurnaceGolem as well as the stomp weapon.
RemembranceOfDancingLion (extends TradableItem)	A class that represents a RemembanceOfDancingLion TradableItem Responsibilities: 1. Extends TradableItem abstract class to provide a concrete example of a TradableItem. 2. Grants whoever has this item the capabilities of a DivineBeastDancingLion as well as the divine powers.
TradeAction (extends Action)	A class that represents a Trading Action by the Player with Suspicious Trader Responsibilities: 1. Allow Player to perform trading with the SuspiciousTrader 2. Extends Action abstract class to perform execute method
StompWeapon (extends WeaponItem)	A class that represents a stomp weapon Responsibilities: 1. Extends WeaponItem as it can be used to attack Actors as well as being picked up or dropped as an item 2. Contains Golem's stomp but as a weapon so that other actors who obtain this item can use golem's stomp as well
FurnaceGolem (extends Enemy)	A class that represents a FurnaceGolem actor. Modifications: 1. Removed capabilities and added FurnaceEngine item into the Golem's inventory

FurnaceEngine (extends StompWeapon)	A class that represents a FurnaceEngine weapon. Responsibilities: 1. Extends StompWeapon class to override allowableActions method to allow actors to use StompAction as long as the other actor is not the SuspiciousTrader. 2. Contains all of the golem's capabilities such as fire_resistance, ability to cause a ring of fire as well as shockwave.
DivineBeastHead (extends DivineWeapon)	A class that represents a DivineBeastHead weapon. Responsibilities: 1. Extends DivineWeapon class to override attack method to allow actors to use divine powers as long as the other actor is not the SuspiciousTrader. 2. Contains all of the DivineBeastDancingLion's abilities and divine powers as well as the function to transit one divine power to another.

<u>Solution and Explanation:</u>

Solutions:

1. SuspiciousTrader
   a. SuspiciousTrader is a class as well as an npc character that extends the Actor class. It does not extend Enemy class as other actors are not able to perform any attack action on this actor. On every tick where there is no trade action being performed, this actor performs DoNothingAction due to its NPC characteristic where it should not move around randomly.
   b. The hitpoint given to this actor is set to 0 with the logic that NPC characters do not have a health bar and will not be able to receive any damage.
   c. No hitpoint checker is given to SuspiciousTrader hence, preventing the execution of the unconscious method that is inherited from the Actor class. This shows the immortality of SuspciousTrader actor.

2. Remembrances
    a. Two tradable items also known as RemembranceOfFurnaceGolem and RemembranceOfDancingLion are introduced and whenever Player trades them, it is given access to FurnaceGolem’s capabilities and powers as well as DivineBeastDancingLion’s divine powers.
    b. A Tradable interface is created with the intention that some items that could be weaponitems or consumables might have the ability to be traded as well.
    c. Trading these two remembrances will give the Player some benefits such as healing effect as well as a buff to its hitpoints and mana.

3. FurnaceEngine & DivineBeastHead
    a. Once the remembrances are traded, the Player is given access to the Golem and DancingLion’s powers, however their powers are coded in their own actor classes respectively.
    b. To overcome the issue, two new items are introduced: FurnaceEngine and DivineBeastHead. FurnaceEngine allows the Player to add new capabilities such as fire resistance, able to perform shockwave as well as a ring of fire around it. The player has the option to use the stomp action on its opponent as well. DivineBeastHead allows the player to gain access to divine powers as well as transition from one divine power to another.
    c. However, this is code redundancy as the exact implementation is implemented twice: once in FurnaceGolem itself and once in FurnaceEngine. Hence, the capabilities of Golem are removed and FurnaceEngine items are added to FurnaceGolem’s inventory. This solves the issue of redundancy.

Advantages:

1. SOLID Principles

    Single Responsibility Principle (SRP)

- The SuspiciousTrader class adheres to SRP as the sole responsibility of this actor is to perform trade action as it does not have any wandering behaviour nor attack actions.

    Open-Closed Principle (OCP)

- With the TradableItem abstract class, all tradable items that extend from this abstract class will inherit a trade method where different functionalities for different items can be implemented and overridden. This prevents modification of the abstract class, hence adhering to OCP.
- By creating FurnaceEngine and DivineBeastHead items where both extend StompWeapon and DivineWeapon weapon items respectively, future actors can obtain these items just by adding them into inventory, thus preventing code redundancy.

    Liskov Substitution Principle (LSP)

- The TradableItem abstract class, implementing the Tradable interface, ensures that any item you create (Remembrance of FurnaceGolem, Remembrance of DancingLion) can be treated as a Tradable.
- By ensuring that TradableItem and any concrete subclasses maintain the behaviors promised by Tradable, they can be substituted freely without breaking functionality. This indicates that LSP is adhered.

    Interface Segregation Principle (ISP)

- The Tradable interface defines only trading-related behaviors, which ensures that classes implementing it such as TradableItem subclasses, are not forced to take on unrelated responsibilities.
- This interface segmentation makes the code more modular, with classes only implementing the functionality they need, fully aligning with ISP.

Dependency Inversion Principle (DIP)

- By using the Tradable interface and the TradableItem abstract class, your Trader class and other high-level modules can interact with items through abstractions rather than concrete classes.
- Similarly, FurnaceEngine and DivineBeastHead are not directly dependent on specific actors or items but can be used by any class that needs their powers. This flexible, decoupled design allows other actors to interact with these power items without depending on the concrete classes where the powers originated.

2. Connascence

   The implementation for REQ2 aims for low Connascence and does so in the following:

   - Connascence of Name (CoN): As trade action class is introduced, it becomes the central place where Tradable and TradableItem names are used. This isolates the name dependencies so that if we rename Tradable or TradableItem, we only need to update TradeAction and any other classes directly interacting with tradable items, not the SuspiciousTrader class. This isolates CoN to TradeAction and reduces its spread across other classes, making renaming and refactoring easier. (Strength: Weak)

   - Connascence of Type (CoT): The SuspiciousTrader interacts only with the abstract type, Tradable, rather than a specific item, reducing CoT dependencies on specific concrete classes. The power weapons (FurnaceEngine and DivineBeastHead) are also handled as types that any actor could acquire, meaning other actors can interact with these power weapons through their specific interfaces or abstract classes, further limiting CoT. (Strength: Weak)

- Connascence of Execution (CoE): No presence of CoE as Actor only obtains the FurnaceEngine and DivineBeastHead after trading both remembrances and this happens in a specific order. It is impossible for the order to be reversed where the Player is able to obtain the weapon items without trading first. (Strength: Low)

Hence, we observe that this implementation indeed has low connascence and hence, low coupling between methods and classes.

3. DRY Principle
   - Powers from FurnaceGolem and DancingLion are extracted into FurnaceEngine and DivineBeastHead classes, respectively. This move encapsulates power-related logic into dedicated power weapon classes, rather than duplicating power functionality in each actor that might use it.

   - With this setup, any actor that needs to access FurnaceGolem's or DancingLion's powers can simply use FurnaceEngine or DivineBeastHead, avoiding the need to reimplement similar power-related methods in each new actor.

   - Any item that can be traded inherits from TradableItem, which avoids duplicating trade-related code in each individual item. For example, both Remembrance of FurnaceGolem and Remembrance of Dancing Lion share trade functionality without requiring repeated code in each concrete class.

4. Code Smell

   <u>Brain Class</u>

   - By creating specific classes for each power weapon (FurnaceEngine and DivineBeastHead), each weapon encapsulates its own powers rather than overloading the FurnaceGolem or DancingLion actor classes. This keeps the actors focused on their roles and avoids turning them into Brain Classes.
   - If FurnaceEngine or DivineBeastHead were to grow highly complex, containing different modes or conditions for each actor, they could risk becoming Brain Classes.

   <u>Intensive Coupling</u>

- Due to the implementation of Tradable interface and TradableItem abstract class, intensive coupling is avoided as the different functionalities are kept in the item's trade method that they inherited from the abstract class, hence no matter how different is each tradable item from each other, it generally does not affect the parent class as well as the interface.

<u>Long Method</u>

- Since each power weapon class (FurnaceEngine, DivineBeastHead) is responsible for its own powers, methods in FurnaceGolem and other actors remain focused on actor-specific logic. This keeps methods in actors concise and clear. If a power weapon ends up with a single method to handle multiple complex powers, it might become lengthy, causing it to be a long method.

<u>Shotgun Surgery</u>

- Shotgun surgery is avoided in the instance of the creation of FurnaceEngine and DivineBeastHead. This is due to singling out all of the capabilities and powers into a single class and allowing actors that are not FurnaceGolem or DivineBeastDancingLion gaining the access by obtaining the power items.

5. Extensibility
    - Due to the fact that the Tradable interface is created, this shows extensibility as this does not only limit the trading method to tradable items. For example, a consumable item that can be consumed can also be traded given that the item implements Tradable interface. This gives the Player options to either consume the item or to trade the item for different benefits.
    - Besides that, the TradableItem abstract class introduces extensibility as all tradable items that extend this abstract class will be able to use the trade method inherited from the parent class.

Possible Limitation/Drawback

- As there is only one NPC actor in the game (SuspicousTrader) for the time being, there is no need for an NPC abstract class. However, in the future, for extensibility purposes, there may be more NPC characters with various behaviours such as one with wandering behaviour which is different from SuspiciousTrader, even though both of them cannot be attacked by Player and other Actors.

- Besides that, in future where more bosses characters drop their respective remembrances, this can cause an issue regarding code redundancies, especially when all of the remembrances' display characters are the same.

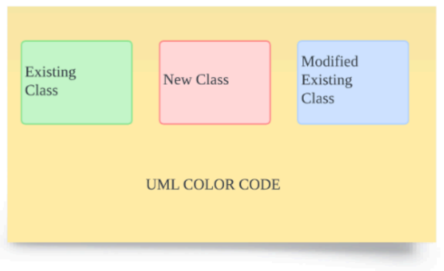
Alternative Solution

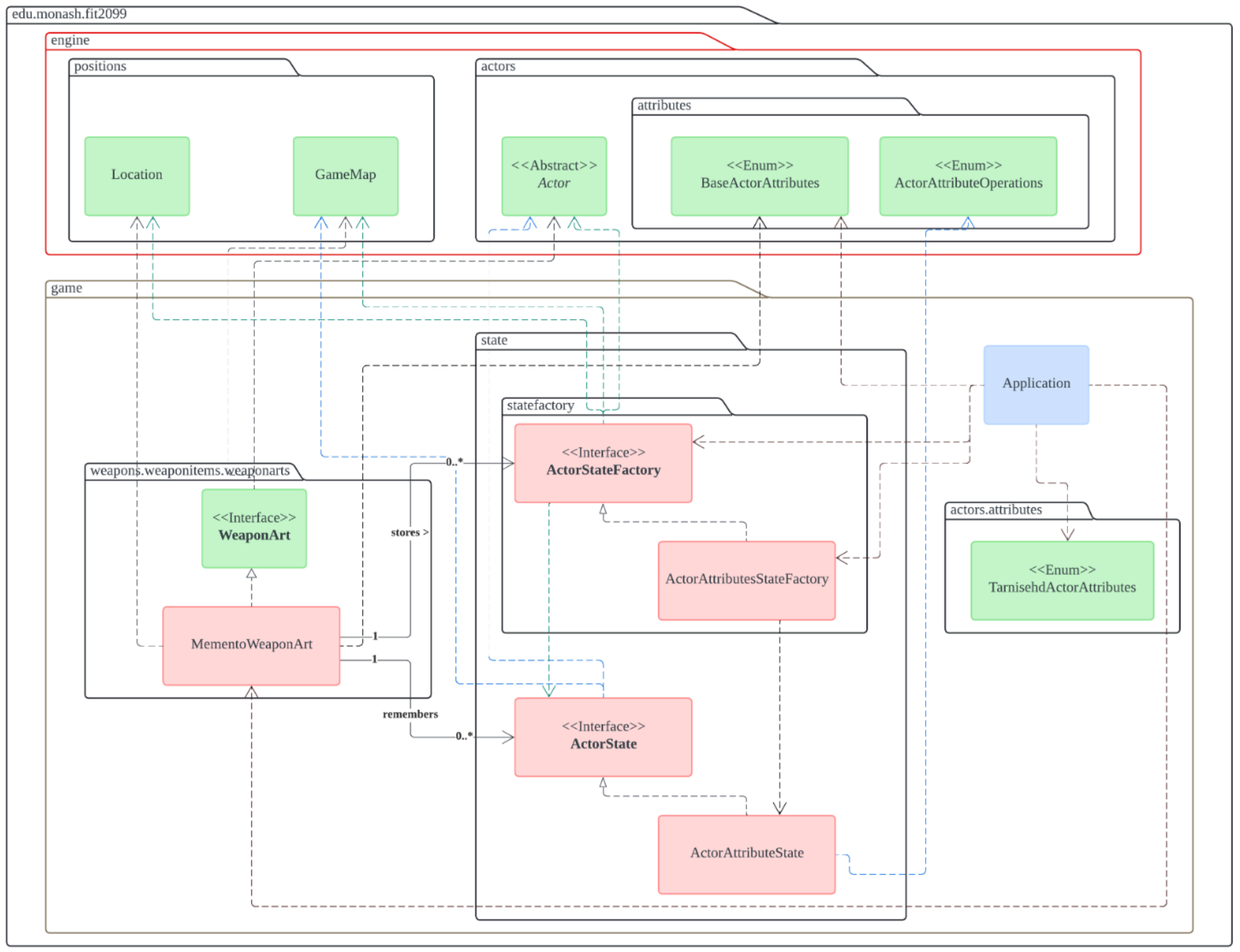
- An NPC abstract class can be created to resolve this issue where the commonalities are certain behaviours such as wandering behaviour or doing nothing until the Player is around the NPC actors. By having an abstract class, this promotes extensibility and reduces code redundancies, preventing violating OCP.

- A remembrance abstract class can also be created when there are more remembrance items created in the game. Commonalities such as their display characters will only be written once in the abstract class to prevent code redundancy. As the current implementation of remembrances inherits TradableItem's trade method where each of these remembrances can override to fit their respective functionalities hence it works as an alternative solution other than creating an abstract class.

Req 3: Memento

Req 3 UML Class Diagram





Reference:

- In case some UML arrows aren't visible, refer to the .png file in Gitlab or click here for the source diagram
- For Sequence Diagram, refer to the .png file in Gitlab or click here for the source diagram

Req 3 Design Rationale

Classes Created / Modified	Roles and responsibility
MementoWeaponArt class (implements WeaponArt) - created	Responsibility: 1. Represents a special ability that a WeaponItem can possess, which is storing and restoring past states of an actor Relationship: 1. Implements the WeaponArt interface to execute the Memento weapon art
ActorStateFactory interface - created	Responsibility: 1. An interface for creating an abstract factory of an actor's state. 2. An actor's state may contain different data values that are independent of one another, like ENUMs, locations, wallet, etc
ActorAttributesStateFactory class (implements ActorStateFactory) - created	Responsibility: 1. A concrete class that creates the state of an actor that can take in all possible ENUM attributes. 2. Implements the ActorStateFactory interface to create the state of an actor's multiple enumerable attributes.

ActorState interface - created	Responsibility: 1. An interface for the state of an actor. 2. Abstract method will restore the state of the actor.
ActorAttributeState class (implements ActorState) - created	Responsibility: 1. A class that represents the state of an actor's ENUM attributes. 2. Implements the ActorState interface to store and restore the state of an actor's enumerable attributes.
Application - modified	Responsibility: 1. Creates a list of enum attributes to be remembered in the Weapon art. 2. Initialises state factories for each different type, such as integers, strings, enums or objects. 3. Zip the factories up together in a list, then put them as arguments into the MementoWeaponArt

<u>Solution and Explanation:</u>

Problem:

1. Need to create a flexible mechanism to save and restore the state of an actor, including attributes and location.
2. Ensure the solution is extensible to accommodate future state properties without modifying the core logic.
3. Implement the Memento pattern to remember and restore the actor's state.
4. Integrate the new functionality into the existing game mechanics.

Solution:

1. Create an ActorState interface to represent the state of an actor.

2. Implement concrete classes for different state properties, such as ActorAttributesState and ActorLocationState.
3. Use the Abstract Factory pattern to create state objects dynamically.
4. Implement the MementoWeaponArt class to save and restore the actor's state using the Memento pattern.

Explanation:

1. Create an ActorState Interface

   First, we define an ActorState interface to represent the multiple states' of an actor. This interface includes a method to restore the state of the actor. This abstraction allows us to define different types of states without modifying the core logic. Examples of states are ENUM values of an actor, the Location of an actor, the actor's Wallet, any current StatusEffect, etc. The reason ActorState was implemented as an interface instead of an abstract class is because the sole purpose of this Interface is to carry the "promise" of implementing the *restoreState* void method. An abstract class would only be more suitable, if it was required that some attributes or values belong to the abstract class, perhaps state expire duration, state cost, or similar. Since ActorState does not require saving any base attributes, it is thus more efficient to make it as an Interface.

2. Implement Concrete State Classes

   We implement concrete classes for different state properties. For example, ActorAttributesState stores the actor's attributes, all of which are currently just ENUM classes. These classes implement the ActorState interface and provide specific logic to restore the state, by storing the attributes initially in (enum, integer) maps and pushing them into the memento stack, and them popping them off sequentially to refer to these past attributes when restoring the player to a previous state.

3. Use the Abstract Factory Pattern

   To make the solution extensible, we use the Abstract Factory pattern. We define an ActorStateFactory interface and implement concrete factories for each state type, like ActorAttributesStateFactory. This allows us to add new state properties without modifying the core logic of the Memento class. For example, including a Location state into our MementoWaponArt can simply be done by creating another concrete Factory class that stores and restores a Location object, e.g. `ActorStateFactory locationFactory = new ActorLocationStateFactory();`

4. Implement the MementoWeaponArt Class

The MementoWeaponArt class uses the Memento pattern to save and restore the actor's state. It maintains a stack of mementos maps and uses the factories to create state objects dynamically. This design ensures that the solution is flexible and extensible, whereby more “states” to be remembered can be easily appended to the list of stateFactories that will make up the remembered 'state' of the actor, and then pushed into the stack.

5. Example Usage

   Finally, the Application class demonstrates how to use the MementoWeaponArt class in the Application class. This is done by first initialising state factories for the desired state properties and passing them as a list into the MementoWeaponArt constructor.

Advantages:

1. SOLID Principles

   Single Responsibility Principle (SRP)

   - MementoWeaponArt adheres to SRP by focusing solely on the creation and restoration of actor states, by pushing and popping previous ‘states’ from the memento stack. For creation of states, the MementoWeaponArt createMemento will create an observable stream of stateFactories, whereby each factory will be in charge of producing a ‘State’ of the user. It does not need to know how the factories generate their respective states.
   - Each class that implements ActorState will handle specific aspects of the actor's state, ensuring clear separation of concerns.

   Open-Closed Principle (OCP)

   - The use of the ActorStateFactory interface allows for the addition of new state types without modifying existing code in the WeaponArt createMemento method , adhering to OCP of this class.

   Liskov Substitution Principle (LSP)

   - ActorState interface ensures that any class implementing it can be used interchangeably, for example the “for-loop” that iterates for each class that implements the ActorState interface. This allows for parametric polymorphism, whereby different unrelated classes can be invoked in a single loop, since they all implement the same ActorState interface and can call the restoreState method.

<u>Interface Segregation Principle (ISP)</u>

- The ActorState interface is focused and only includes the restoreState method as it is relevant to state restoration, adhering to ISP.
- Further classes that are responsible for state memory and restoration will need to implement ActorState's restoreState method.

<u>Dependency Inversion Principle (DIP)</u>

- MementoWeaponArt depends on the ActorStateFactory abstraction rather than concrete implementations, ensuring that the code blocks for createMemento and restoreMemento are short and concise.

2. Connascence

The implementation for REQ3 aims for low Connascence and does so in the following:

<u>Connascence of Name (CoN)</u>

- Methods like createMemento and restoreMemento rely on their names being consistent across the system.
- This only causes mild CoN since MementoWeaponArt is currently the only class in this program that utilises the properties of ActorState and ActorStateFactory. (Strength: weak).

<u>Connascence of Type (CoT)</u>

- The ActorStateFactory interface and its implementations must consistently use the ActorState type. This ensures that each factory can produce its respective state type, which is also what we want. (Strength: weak).

<u>Connascence of Meaning (CoM)</u>

- The interpretation of actor attributes through the use of ENUMs must be consistent across the system. For example, BaseActorAttributes like HEALTH, MANA as well as TarnishedActorAttributes like STRENGTH will be stored as key value pairs in a Map<Enum<?>, **Integer**>. This indicates that health, mana and strength values need to be the same type, in this case integers, not float values.
- This connascence should not be ignored, but likewise, should not pose too much of a problem due to Java's *Math.round* function to convert float values to integers without downcasting. (Strength: moderate).

Hence, we observe that this implementation indeed has low connascence and hence, low coupling between methods and classes.

3. DRY Principle
   - The ActorState interface and its implementations reduce code duplication by encapsulating state-related logic in dedicated classes, reducing code duplication for each concrete State class that implements restoreState method from the ActorState interface.
   - The MementoWeaponArt class centralises the logic for saving and restoring actor states, avoiding repetitive code and ensuring the purity of other weapon art classes through the use of the WeaponArt interface.

4. Extensibility
   - The ActorStateFactory interface allows for easy addition of new state types, making the logic of MementoWeaponArt and all its related systems extensible. All the user needs to do is create new Factories, which will then be automatically accounted for in the createMemento stream and the restoreMemento polymorphic for-loop.
   - New state properties can be added by creating new implementations of ActorState and ActorStateFactory without modifying existing code, for example creating a ActorLocationState to store the player's previous location, or a WalletState, to store previous amounts of money.
   - Since the restoreState method of the ActorState interface takes the parameters of both the attacker and the target Actor, our code now becomes possible for future implementations of Memento to be used on other entities (other than the wielder). For example, "Memento" may be used against the target of the attack, allowing it to remember the target's health and status effects. When the restoration step is triggered, the target's status effects will be restored to its previous state, by simply swapping the argument of Actors "attacker" with the argument "target".

5. Code Smell

    Brain Class

    - No class currently exhibits brain class characteristics, but MementoWeaponArt could become one if it starts handling more responsibilities without the help of newly created stateFactories.

    Intensive Coupling

    - MementoWeaponArt and ActorStateFactory are tightly coupled, but this is necessary for the current design, since memento assigns all responsibilities of remembering past actor states to the factory classes.

    Long Method

    - Methods in MementoWeaponArt are concise, but if the logic for state creation or restoration becomes more complex, they could become long methods.

    Shotgun Surgery

    - Changes to the state logic (for example, the createState method in ActorStateFactory) would require modifications in multiple classes (MementoWeaponArt and ActorState), indicating potential for shotgun surgery. However, the chances of this happening is low, as our current implementation of the createState method already accounted for possible future modifications, though the inclusion of all three extensively-used parameters (Actor actor, Location location, GameMap map).

Possible Limitation/Drawback

As mentioned in the Brain Class code smell, the logic for creating and storing different 'states' of one memento in MementoWeaponArt may be seem as troublesome, as the user needs to create both a new State class, as well as a Factory class that generates it, essentially enforcing 2 new classes per state, instead of one

Alternative Solution

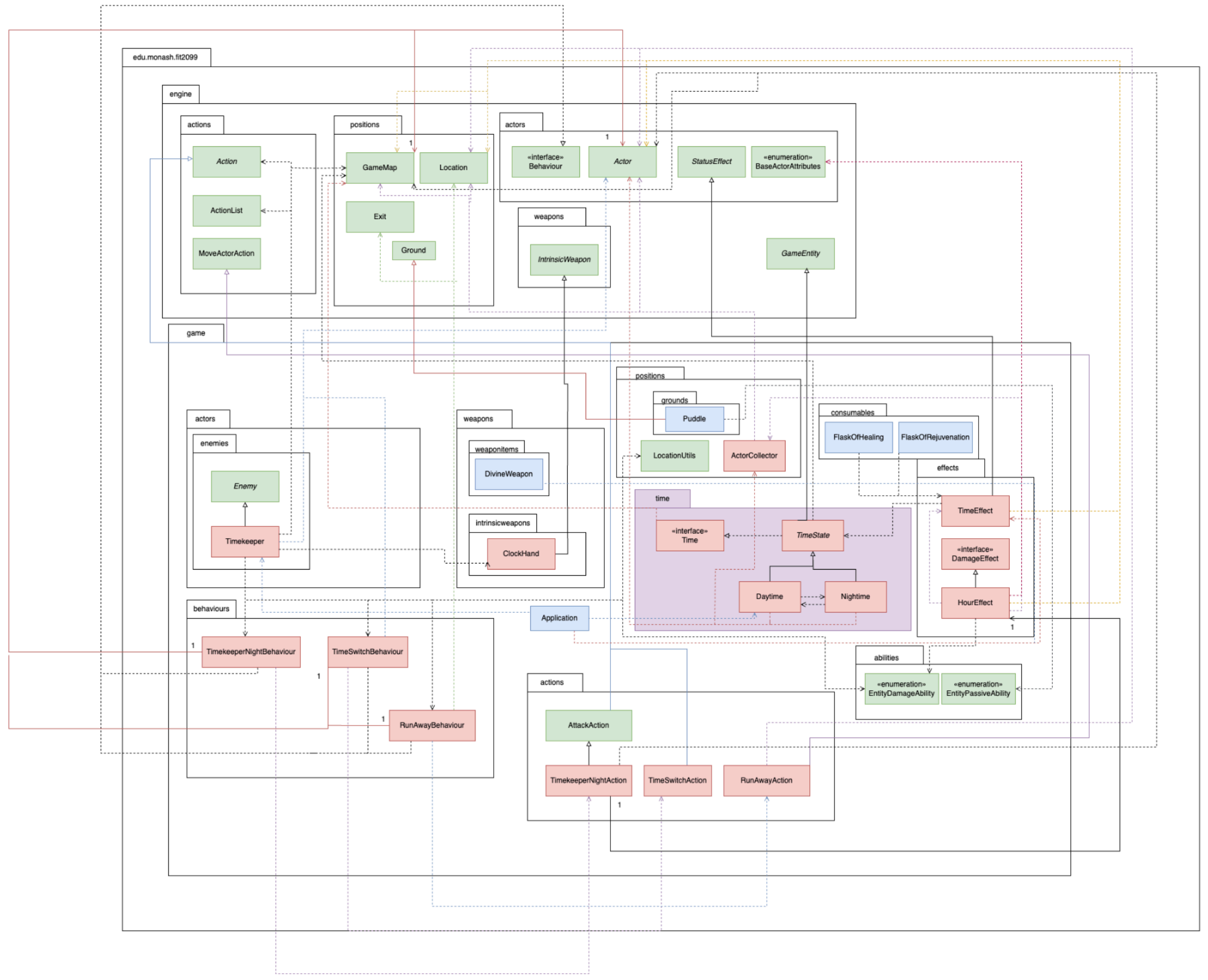
Combine the responsibility and abstract method calls of both ActorState and ActorStateFactory interfaces into one single interface. This means that each new state only requires creating one new class, instead of creating one pair of classes (state class, state factory). This single interface will now contain 2 (previously only 1) abstract methods that must be implemented by future state classes.

At first glance, this might seem like a good modification. However, keep in mind that the parametric-polymorphism properties used in the MementoWeaponArt will no longer be viable. This means that slight modifications have to be made in the MementoWeaponArt each time we want to create a new 'state'.

Evaluating the pros and cons of our current design and the alternative solution, it is clear that creating a pair of (state class, state factory) might seem tedious, but it actually brings more benefits and creates less OOP disadvantages compared to the alternative solution, especially if the number of states were to be scaled to arbitrary amounts in the future.

Req 4: CREATIVE MODE

Req 4 UML Class Diagram



Colour code

🟩 Unmodified, existing class	🟥 New Class
🟦 Modified, existing class	

Req 4 Design Rationale

Classes Created / Modified	Roles and responsibility
Timekeeper - created	Represents the Timekeeper boss - Runs away from the Player at day - May change the time into Night when player is within its surroundings - At night, it will follow and attack the Player using its intrinsic weapons: SecondHand and MinuteHand - Attacks every entity globally using the HourEffect every 24 ticks Relationships - extends Enemy
ClockHand - created	Represents the Timekeeper's ClockHand - Has a 10% chance of stabbing the Player with MinteClockHand, dealing 30 damage - Has a 50% chance of stabbing Player with SecondClockHand, dealing 5 damage Relationships - extends IntrinsicWeapon
RunAwayAction - created	A class that represents an action where an actor runs away from another actor Relationships - Extends MoveActorAction

RunAwayBehaviour - created	Factory to create the RunAwayAction Relationships - implements Behaviour
TimeSwitchAction - created	A class that represents the TimeSwitch action - There is a 15% chance to trigger this action when the player is within the surroundings of the Timekeeper Relationships - Extends Action
TimeSwitchBehaviour - created	Factory to create the TimeSwitchAction Relationships - Implements behaviour
TimekeeperNightAction - created	A class that represents the TimekeeperNight action - Calls the hour effect to deal the HourEffect damage onto all entities Relationships - Extends AttackAction
TimekeeperNightBehaviour - created	Factory to create TimekeeperNightAction Relationships - Implements behaviour
ActorCollector - created	Collects all the actors in the game

HourEffect - created	Represents the HourEffect dealt by the Timekeeper, which deals a global damage of 30% of each actors health - Happens every 24 ticks Relationships - Extends DamageEffect
Time (interface) - created	An interface that represents the time in the game - Enforces 3 methods: applyTimeEffects, transition, and getCurrentTime
TimeState (abstract class) - created	A class that represents the state of time in the game Relationship - Implements Time
Daytime - created	A class that represents the Daytime state of the game - Restores all flasks to normal charges - Enemies deal normal damage Relationship - Extends TimeState
Nightime - created	A class that represents the Nightime state of the game - All flask charges are halved - Enemies deal double damage Relationship - Extends TimeState
Puddle - modified	Added a new drowning mechanism - 5% chance to drown drownable entities

	Relationships - Extends Ground
FlaskOfHealing - modified	Modified to update charge count with regard to TimeStates
FlaskOfRejuvenation - modified	Modified to update charge count with regard to TimeStates
DivineWeapon - modified	Modified to update damage multiplier with regard to TimeStates
EntityDamageAbility - modified	Added new damage abilities: *HOUR, MINUTE, SECOND*
EntityPassiveAbility - modified	Added new passive ability: *DROWNABLE*
Application - modified	Added Timekeeper into Gravesite plain maps, initialised the TimeEffect

Solution and Explanation:

Problem:

- Create a new boss, Timekeeper. The Timekeeper has two intrinsic weapons: MinuteClockHand, and SecondClockHand. Additionally, The Timekeeper is capable of changing the time into Night. The Timekeeper only attacks the Player at night. At day, it will run away from the player. At night, it will follow and attack the Player. Lastly, for every 24 ticks that the Timekeeper is alive, it will deal a global 30% damage on every actor's health.
- Integrating a new day and night cycle into the game, with 24 ticks representing one day, and 12 ticks for both day and night.
- At day, enemies deal normal damage onto the player, and all flasks have their normal charges
- At night, enemies double damage and flask charges are decreased to half

- Make Timekeeper drownable, and have the ability to enter floors

Solution:

- To simulate the passage of time, we use a TimeEffect class that manages the time across all maps
- The player will have this status effect throughout the game
- Create a new class that collects actors both across all maps and specific maps to simulate the HourEffect
- Creating Daytime and Nightime classes to handle logic of both TimeStates in specific classes
- Create Actions for all the attacks the Timekeeper is capable of doing, and its corresponding behaviour classes
- Use methods in TimeEffect to communicate the timing to Flasks, ensuring that they have the correct charges at day and night

Explanation:

Consistency and Scope of TimeEffect

- The TimeEffect class manages synchronising all maps, which ensures consistent day-night transitions and keeps actors and items affected uniformly across the game. By centralising time management, TimeEffect avoids issues where different maps might desynchronize and ensures that any time-based effects apply to all areas without separate tracking per map.
- If additional time-based effects (like weather or seasons) are introduced, the current structure could either extend TimeEffect directly or introduce new classes to maintain separation of concerns. Creating a class for specific new effects would prevent overloading TimeEffect and keep responsibilities focused.

Separation of Daytime and Nighttime Behaviours

- The Daytime and Nighttime classes handle actor and item effects specific to each time phase, ensuring that each time-of-day behaviour is encapsulated and doesn't interfere with others. This separation supports flexibility, such as the night phase triggering double damage, while making it simple to add or modify effects specific to each phase.
- If new time phases like “dawn” or “twilight” are added, you could extend the system by creating additional classes like DawnPhase or TwilightPhase. This modular approach would align with SRP and OCP, as each phase’s logic would remain independent, while the TimeEffect class could coordinate which phase is currently active.

Global Actor Collection and HourEffect Design

- To maintain performance, global actor collection is implemented efficiently in HourEffect, collecting actors across maps in a controlled way. This approach ensures that the Timekeeper can apply global effects, like its periodic damage, without causing significant delays.
- For cases where certain actors need to be exempt from global effects, such as actors in specific regions or under special effects, the system could extend to include conditions for actor eligibility within HourEffect. This would add flexibility while avoiding unnecessary processing for actors that shouldn't be impacted by the global effect.

Timekeeper Action Modularity

- By implementing Timekeeper's actions (such as RunAwayAction for daytime and AttackAction for nighttime) as distinct classes, the design is modular and adheres to SRP. Each action is responsible for a specific behaviour, allowing easy modification or extension without changing the Timekeeper itself.
- If new behaviours are added, like additional nighttime attacks or different daytime actions, the modular design supports adding new actions or modifying the Timekeeper's behaviour sequence without affecting existing code. This allows flexibility, so the Timekeeper can evolve with new features while adhering to the open/closed principle.

General Extensibility and Modularity

- The current design supports adding new actor types or game elements by letting time-based effects apply generally, while specific behaviours can be customised within Daytime and Nighttime as needed. For example, additional actors like a "Solar Entity" could interact with the day-night cycle by adding unique actions to Daytime or Nighttime that are only executed for that actor type.
- If future mechanics introduce additional global effects dependent on time, the existing TimeEffect class could either be extended with new methods or work alongside a parallel system (e.g., WeatherEffect). This approach would prevent TimeEffect from handling too many unrelated responsibilities, maintaining its primary focus on time simulation while allowing new global effects to function independently.

Advantages:

SOLID Principles

SRP (Single Responsibility Principle)

- Each class in this design addresses a single responsibility. The TimeEffect class manages the game's time progression, updating ticks and determining the transition between day and night. Daytime and Nighttime classes handle specific behaviours, such as halving flask charges at night and returning them to normal during the day. This separation of concerns ensures each class handles one specific concept: TimeEffect manages the passage of time, Daytime and Nighttime focus on time-based behaviour, and the Timekeeper class manages boss behaviours based on time. For example, during the day, the Timekeeper switches to a RunAwayAction and, at night, switches to a FollowAction. The RunAwayAction and FollowAction classes each handle a distinct behaviour, so any adjustments to one behaviour (like adjusting the range for RunAwayAction) don't affect the other, simplifying future changes and debugging.

OCP (Open/Closed Principle)

- This design supports easy extension with minimal changes to existing code. If a new time-based phase, like Twilight, were to be added, only a new class for the phase, like Twilight, would need to be implemented. The existing TimeEffect class would require no modifications, as it's designed to operate with any state that implements the same behaviours. Similarly, if Timekeeper gained new behaviours, these could be added by creating new action classes without changing existing behaviours, making it simple to extend the system while preserving existing functionality.

LSP (Liskov Substitution Principle)

- The design aligns with LSP, allowing subclasses to extend from time-based states like Daytime and Nighttime without causing errors. If a new state like Twilight is introduced, it can be treated just like other time phases, as it would share the same interface or abstract class, such as TimeState. For example, Twilight could implement specific effects (e.g., granting players a speed boost) without requiring changes to the classes that rely on time-based transitions, allowing flexible use of new states in gameplay logic.

ISP (Interface Segregation Principle)

- Classes in this design only implement methods and functionalities necessary for their purpose. The Daytime and Nighttime classes specifically manage time-based logic, handling the effects and transitions associated with their respective phases without needing to take on additional responsibilities. By avoiding extraneous methods in these classes, the design ensures that each class remains streamlined, focusing solely on the relevant time-based behaviours without incorporating unrelated game mechanics, thereby keeping the codebase cleaner and more manageable.

DIP (Dependency Inversion Principle)

- The design adheres to DIP by structuring dependencies around abstractions rather than concrete implementations. High-level components, such as the game world or Timekeeper behaviours, interact with abstract classes like TimeEffect and TimeState, allowing flexibility. For instance, if the way time is managed needs modification, updates are made only in TimeEffect without impacting classes that rely on time states, such as Timekeeper or player behaviours. This separation reduces the impact of changes, as the larger system depends on the abstraction of time progression rather than a specific implementation, allowing for simpler adjustments in the future.

Connascence Reduction

Connascence of Name

- Connascence of Name is reduced by encapsulating day and night behaviours within the Daytime and Nighttime classes. For example, Daytime includes behaviour for daytime actions like RunAwayAction for the Timekeeper, while Nighttime manages actions such as FollowAction or reducing flask charges. By separating these behaviours into distinct classes, changes to daytime actions don’t unintentionally impact nighttime behaviour and vice versa. This prevents issues where renaming or modifying one behaviour could accidentally propagate to others, reducing naming dependencies.

Connascence of Position

- The design minimises Connascence of Position by organising time states and behaviours within distinct classes rather than relying on positional arguments or arrays to define day/night transitions. For instance, instead of modifying specific positions within a data structure for time-dependent behaviours, Daytime and Nighttime classes manage these internally. This means updates to day or night behaviours don’t involve adjusting positional values in unrelated classes. Additionally, Timekeeper simply references TimeEffect for current time without needing positional data, avoiding dependencies on positional ordering, which could cause errors if updated incorrectly.

Connascence of Meaning

- Connascence of Meaning is reduced by assigning specific responsibilities to Daytime, Nighttime, and TimeEffect, giving each class a well-defined role. For example, TimeEffect exclusively tracks time progression and provides a straightforward day/night state interface, while Daytime and Nighttime focus on behaviour corresponding to each time state. This clear separation ensures each class has a distinct and meaningful role—Daytime never manages night effects, and Nighttime avoids tracking time itself. This clarity makes updates and future extensions less error-prone, as the meaning of each class remains unambiguous.

Connascence of Type

- In this design, Connascence of Type is addressed by relying on polymorphism and encapsulation rather than hard-coding specific types or direct type dependencies. For instance, the TimeEffect class doesn't directly manage Daytime or Nighttime behaviours but rather exposes time-related information. The Timekeeper class then interacts with Daytime or Nighttime classes based on the current game time without needing to know the specifics of either. This setup reduces type dependencies across classes, so changing one time-state class wouldn't directly impact the others or the TimeEffect class, helping avoid potential issues from type-based connascence.

Connascence of Execution

- Connascence of Execution, which concerns the order in which operations occur, is minimised by handling distinct behaviours within separate classes and managing action priorities within a TreeMap. For instance, actions like RunAwayAction (during the day) and FollowAction (at night) are organised in a way that each time state triggers the appropriate behaviour based on the game tick. This reduces the likelihood of unintended behaviour caused by changes in action order since each behaviour is managed independently within the Daytime and Nighttime classes. By relying on encapsulated actions and priorities in the TreeMap, execution dependencies are minimised.

Connascence of Identity

- The design limits Connascence of Identity by not requiring specific, fixed instances of Daytime, Nighttime, or Timekeeper to exist across classes. For example, TimeEffect manages time progression independently and only provides a day/night state, allowing Timekeeper to adapt to any instance of Daytime or Nighttime. This means that Timekeeper is not dependent on a particular instance of a day or night manager; it only needs to access the current state from TimeEffect. This makes the design more modular and less vulnerable to issues arising from instance-specific dependencies.

DRY (Don’t Repeat Yourself)

- The design minimises duplication by centralising time-related logic within the TimeEffect class, which manages the global passage of time for all maps, reducing the need to implement similar logic across various locations or actors. This approach allows the game to adjust day/night cycles, apply effects like the 24-tick health reduction, and manage time-based changes in one place. This centralization reduces maintenance complexity, as adjustments to time can be made in TimeEffect without needing updates in each affected class. Additionally, Timekeeper's attack behaviours are encapsulated in dedicated classes such as RunAwayAction, FollowAction, and TimekeeperNightBehaviour. By isolating each behaviour, the design avoids redundant logic, as each class performs a single action, making it reusable across other characters or instances without duplicating code.

Extendability

- The game’s modular design supports easy expansion. For example, if a new character like a “Sunkeeper” were added, designed to operate solely during the day, it could be integrated by adding new behaviours and modifying the TimeEffect class without changing existing classes. The TimeEffect, Daytime, and Nighttime classes are structured to be extendable, allowing new mechanics like weather or seasonal effects to be incorporated within TimeEffect itself, thereby enhancing environmental diversity without modifying the entire system. Additional time phases or events could be introduced by creating new classes like Twilight to manage mid-cycle events. The design also supports HourEffect simulation, where future expansions could include global events triggered at various intervals—like unique power-ups or environmental events that apply across the game world. This ensures the system is ready for new interactions without altering existing behaviours, as the design aligns with OCP principles, keeping existing implementations intact while supporting diverse game mechanics.

Code smells

Brain Class

- TimeEffect only manages the game's day and night cycles, handling basic time-tracking operations and broadcasting time changes, rather than centralising multiple systems' logic. If it began handling complex mechanics like player interactions or combat adjustments outside of time-based effects, it could turn into a brain class. However, the separation of concerns keeps it dedicated to time-related functionality only, making it easy to maintain and avoiding the brain class smell.

Brain Method

- Currently, no method is overburdened with logic to the point of being a brain method. For instance, the method in Timekeeper that manages behaviour shifts between day and night remains focused on transitioning behaviours based on the time, without handling unrelated logic. If this method were responsible for additional processes like controlling Timekeeper's movements, managing health, or handling attack effects, it could start resembling a brain method.

Data Clumps

- Data clumping is effectively managed by keeping time-related effects within the TimeEffect class rather than distributing these variables across multiple classes. For example, Timekeeper accesses time changes through TimeEffect without needing to hold variables like tick count, day/night states, or damage percentages locally. If additional attributes were directly assigned to Timekeeper, such as nightAttackMultiplier or dayDamageReduction, it could lead to clumped data. By centralising time-based data in TimeEffect, the implementation remains cohesive, avoiding data clumps.

Feature Envy

- No class exhibits feature envy as each class remains within its own functional boundaries. For example, Timekeeper interacts with TimeEffect for time-related information without managing or modifying time itself. If Timekeeper included methods that modified time, it would suggest feature envy because it would be assuming TimeEffects' responsibilities. Instead, Timekeeper only adjusts its own behaviours based on time, letting TimeEffect handle time logic independently.

God Class

- The TimeEffect class could risk becoming a god class if it were to take on roles outside of time tracking, such as weather or other game-wide mechanics. For now, however, it is focused solely on managing time-related transitions and state changes. An example of avoiding this smell is how time effects are specific and limited—TimeEffect isn't handling game events, enemy actions, or player state changes, which are managed by separate classes. This single responsibility keeps TimeEffect small and focused.

Intensive Coupling

- The current setup minimises coupling by ensuring that classes like Timekeeper and TimeEffect communicate indirectly rather than having deep dependencies. For example, Timekeeper merely checks the time state in TimeEffect without embedding complex interactions or requiring TimeEffect to be modified to accommodate Timekeeper's logic. If a change were made to the time intervals in TimeEffect, only Timekeeper's method to fetch the state would need slight adjustments, reducing potential coupling issues.

Long Method

- The methods in Timekeeper are kept short. For example, the method handling the switch between day and night actions is limited to checking the time and adjusting Timekeeper's behaviour accordingly. Furthermore, we defined respective actions and corresponding behaviours to ensure this code smell does not occur.

Shotgun Surgery

- There is minimal risk of shotgun surgery if TimeEffect changes because related behaviour changes are isolated. For example, Timekeeper uses the day and night states without tightly coupling to TimeEffect's internal logic, so changes to the time-tracking mechanism (such as switching from tick-based to a different timing system) wouldn't require widespread refactoring. This reduces the chances of needing to modify multiple classes, even with a significant change to TimeEffect.

Type Checking

- The design avoids type checking by using method calls and conditionals rather than direct type comparisons. For example, Timekeeper determines behaviour by checking TimeEffect states (day or night) rather than querying specific time types or using complex type hierarchies. If Timekeeper relied on various subclasses or time type distinctions within TimeEffect, type checking would likely be necessary, but the simplified state-checking approach helps avoid this issue.

Possible Limitation/Drawback

Potential for Increased Complexity

- Introducing multiple classes like TimeEffect, Daytime, Nighttime, and HourEffect adds structural layers that may become challenging to manage if the game doesn't expand with additional time-dependent effects. For example, the TimeEffect class currently governs the day/night cycle and tick-based damage effect. While this centralization is efficient for current requirements, it could lead to a more complex structure than necessary if only a few effects are tied to time, especially if interactions between time states and actors multiply. Additionally, managing time-dependent behaviours across maps could add unnecessary layers of abstraction. For instance, if HourEffect grows to apply various complex effects to all actors globally, debugging interactions between TimeEffect and actors or maps could become difficult.
- Furthermore, if the Nighttime class must manually include every consumable affected by nighttime-specific rules, like halving flask charges, the maintenance burden may increase as new consumables are added over time. This approach can create extra steps for developers to remember to update Nighttime whenever new consumable types are introduced, which could introduce inconsistencies if overlooked.

Performance Overhead

- Applying global effects like HourEffect to all actors across multiple maps could introduce performance issues. Since HourEffect iterates through every actor, applying effects as part of the game's main loop, gameplay could slow down in larger game worlds with many actors. Each tick might involve extensive processing, particularly if effects such as health drains or buffs are applied on each iteration, creating a bottleneck in games with large maps or complex AI actors. Over time, this can lead to noticeable slowdowns or delayed responses for players.

Connascence of Timing

- The time-based design introduces temporal coupling, or Connascence of Timing, since several aspects of the game, such as day/night transitions, health drains, and Timekeeper's behaviours, rely on specific time intervals. For instance, if the tick system in TimeEffect changes to accommodate a different day/night ratio or tick speed, it would require coordinated updates across multiple classes. This could potentially desynchronize elements like Timekeeper behaviours or consumable states, causing issues like premature effects or missed triggers. Ensuring that all components stay in sync with time changes can be challenging as the system grows, increasing the chance of bugs related to timing mismatches.

Alternate solution

The Timekeeper could handle the game's global time system directly, managing the day and night cycle. That is, instead of a separate TimeEffect class, the Timekeeper would update time-related states (ticks, day/night transitions) each time it takes a turn. However, using the Timekeeper to manage the game's global time system directly would create several issues. It introduces a single point of failure, as defeating or removing the Timekeeper would halt time progression. This design violates the Single Responsibility Principle by combining boss behaviour with time management, making it harder to maintain and extend the game. Tight coupling between the Timekeeper and global systems would complicate future changes or additions to time-based mechanics. Additionally, it could lead to inconsistencies in the day/night cycle when the Timekeeper isn't active or present, making the time system less reliable across the game world.

Some advantages of this is that it would centralise time-related logic within a single entity, simplifying how time progression and effects are triggered, especially during Timekeeper interactions. This could enhance immersion, as players would directly associate the passage of time with the presence and actions of the boss. Additionally, it might reduce the need for a separate time management system, potentially streamlining the codebase by keeping time and boss behaviour within the same class.