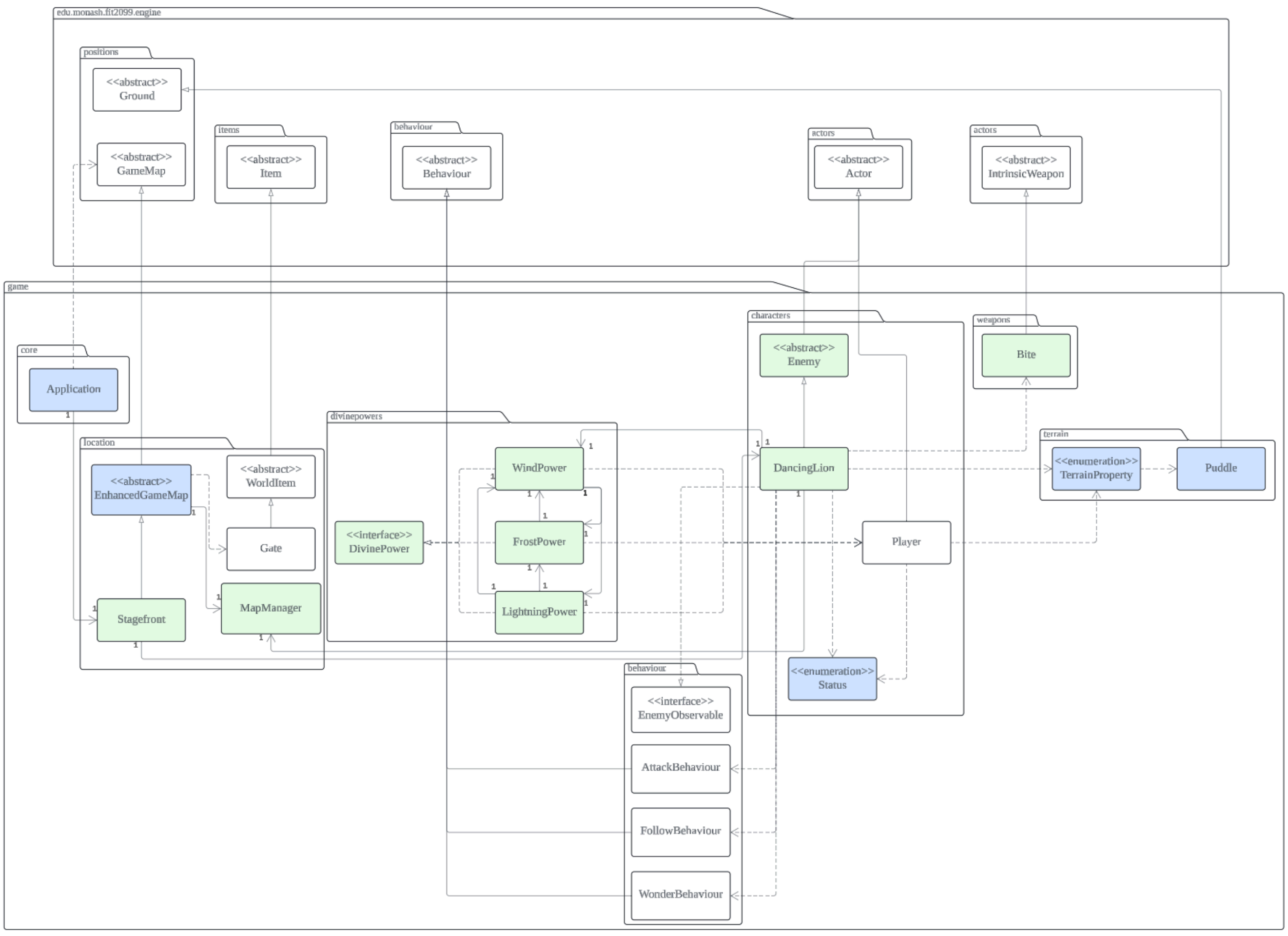
FIT2099 A3 Design Rationale

Group 6 - Ian Lai, Li Junyan, Guo Xun Tan, Fei Yuhang

REQ1

UML



*** new classes are highlighted in Light Green, Modified Existing classes are highlighted in Light Blue*

REQ 1 Rationale

Classes Modified/ Create	Roles and Responsibilities	Rationale
DivinePower(Interface)	Acts as a contract for all divine power implementations, ensuring consistency in special attack behaviour and divine power transitions.	**Alternate Solution**:Define specific classes for each divine power without an interface. **Finalised Solution**: Create a DivinePower interface to ensure all divine powers share a common structure.

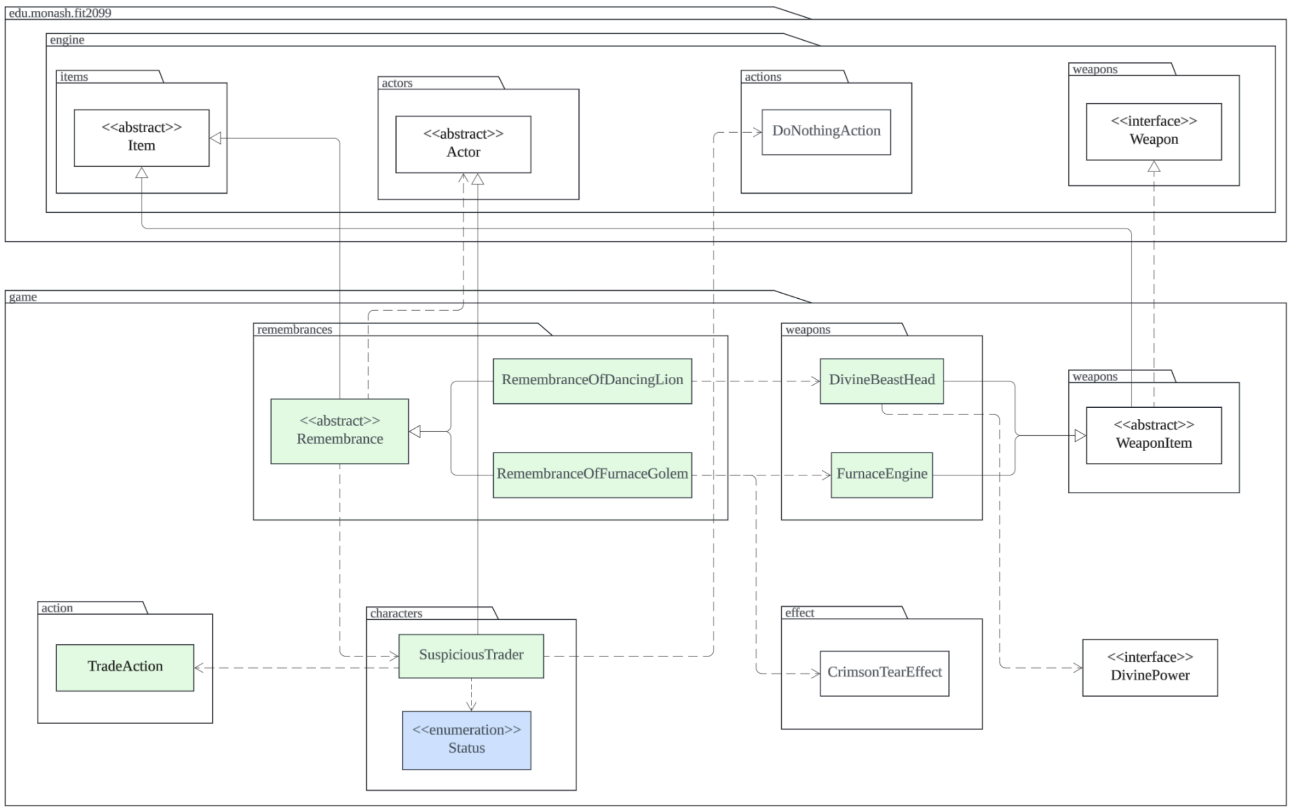
		Reasons for Decision: **LSP:** Allows any divine power to be used polymorphically, simplifying code that utilises these powers. **ISP:** Reduces dependencies by allowing each divine power to implement its own logic for special attacks and transitions. **Limitations and Tradeoffs**: Introduces additional complexity in managing multiple implementations of the DivinePower interface.
FrostPower	Executes a special attack that causes the target to slip and drop all items if standing on water.	**Alternate Solution**: Implement the frost effect as a simple method within the Dancing Lion class. **Finalised Solution**: Create a FrostPower class that implements the DivinePower interface. **Reasons for Decision**: **SRP**: Encapsulates the frost logic within a dedicated class, making it easier to manage and extend. **OCP**: Allows for new divine powers to be added without modifying existing classes. Limitations and Tradeoffs: Increases the number of classes, which may complicate the class hierarchy.
LightningPower	Executes a special attack that creates an aftershock of lightning, damaging all actors in the surroundings	**Alternate Solution**: Implement the lightning effect as a simple method within the Dancing Lion class. **Finalised Solution**: Create a LightningPower class that implements the DivinePower interface. **Reasons for Decision**: **SRP**: Encapsulates the lightning logic within a dedicated class, making it easier to manage and extend. **OCP**: Allows for new divine powers to be added without modifying existing classes. **Limitations and Tradeoffs**: Increases the number of classes, which may complicate the class hierarchy.
WindPower	Executes a special attack that moves the target to a random adjacent location	**Alternate Solution**: Implement the wind effect as a simple method within the Dancing Lion class. **Finalised Solution**: Create a WindPower class that implements the DivinePower interface. **Reasons for Decision**: **SRP**: Encapsulates the wind logic within a dedicated class, making it easier to manage and extend. **OCP**: Allows for new divine powers to be added without modifying existing classes. **Limitations and Tradeoffs**: Increases the

		number of classes, which may complicate the class hierarchy.
Stagefront (Extends GameMap)	Defines the game map for the "Stagefront" scenario, including its layout and initial entity placement.	**Alternate Solution**: Use a generic game map without specific initialization for the "Stagefront" scenario. **Finalised Solution**: Create a Stagefront class that extends GameMap and initialises entities and layout for the scenario. **Reasons for Decision**: **SRP**: Centralises the setup for the "Stagefront" scenario, making it easier to manage and modify. **OCP**: Facilitates changes to the scenario without affecting other parts of the game. **Limitations and Tradeoffs**: Increases the number of map-specific classes, which may lead to higher maintenance effort for map-related code.
DancingLion (Extends Enemy)	Represents the Dancing Lion enemy with unique behaviours and divine powers, providing a dynamic and challenging opponent for the player.	**Alternate Solution**: Incorporate the Dancing Lion's behaviour and divine powers directly within the Enemy class. **Finalised Solution**: Implement the DancingLion class that extends Enemy and incorporates unique behaviours and divine powers. Reasons for Decision: **SRP**: Concentrates the Dancing Lion's specific behaviour and divine power management within a single class. **LSP**: Ensures that the Dancing Lion can be treated as a general enemy where appropriate, while maintaining its unique characteristics. **Limitations and Tradeoffs**: Adds complexity due to the additional class hierarchy and the need to manage divine power transitions and behaviours.
Bite (Extends IntrinsicWeapon)	Models the Dancing Lion's bite attack, incorporating its damage, hit chance, and special effects linked to the current divine power.	**Alternate Solution**: Define the bite attack as a simple method within the Dancing Lion class without inheritance. **Finalised Solution**: Create a Bite class that extends IntrinsicWeapon to model the Dancing Lion's bite attack. **Reasons for Decision**: **SRP**: Encapsulates the bite attack's behaviour, including damage and special effects, within a dedicated class. **OCP**: Allows for future modifications or extensions of the bite attack without affecting other parts of the Dancing Lion's logic. : Increases the number of weapon-specific classes, which may

		complicate weapon management and extension.
Enemy (Abstract Class)	Serves as a base class for all enemy types, encapsulating common enemy attributes and behaviours.	**Alternate Solution**: Implement all enemy types as separate, unrelated classes. **Finalised Solution**: Design an `Enemy` abstract class that serves as a base for all enemy types. **Reasons for Decision**: **LSP**: Ensures that all enemies can be treated uniformly where appropriate, such as in game mechanics that involve enemy interactions. **ISP**: Reduces the complexity for subclasses by providing a common base with shared functionality. **Limitations and Tradeoffs**: May introduce assumptions about enemy behaviour that not all enemy types may satisfy, requiring careful design to avoid overgeneralization.

REQ2

UML



*** new classes are highlighted in Light Green, Modified Existing classes are highlighted in Light Blue*

REQ 2 Rationale

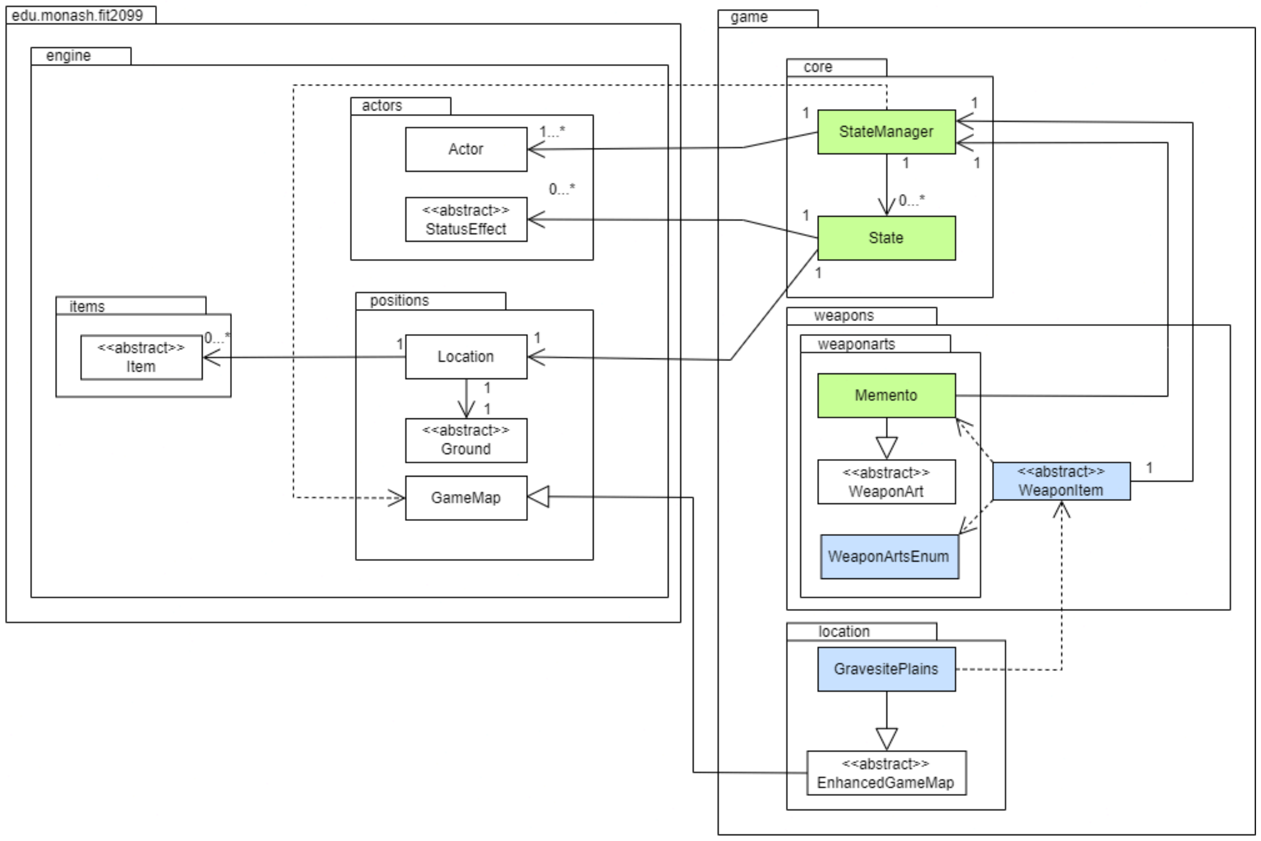
Classes Modified/ Create	Roles and Responsibilities	Rationale
SuspiciousTrader	Represents the non-player character (NPC) Suspicious Trader in the game. Manages the trading interactions with the player, offering unique items in exchange for Remembrances.	**Alternate Solution**:Implement trading functionality within the Actor class. **Finalised Solution**: Create a dedicated SuspiciousTrader class to manage trading interactions. **Reasons for Decision**: **SRP:** Encapsulates trading logic within a specific class, simplifying the Actor class. **OCP:** Allows modifications to trading behaviour without affecting other Actor subclasses. **Limitations and Tradeoffs**: Introduces an additional class, increasing the complexity of the class hierarchy.
TradaAction	Handles the player's decision to trade with the Suspicious	**Alternate Solution**: Handle trading directly within the Actor's turn method.

	Trader. Manages the trading process, including presenting trade options and applying the effects of traded items.	**Finalised Solution**: Create a TradeAction class to manage the trading process. **Reasons for Decision**: **SRP**: Centralizes trading logic, making it easier to manage and extend. **OCP**: Facilitates changes to trading mechanics without impacting other game actions. **Limitations and Tradeoffs**: Adds complexity by introducing a new action class.
Remembrance (abstract)	Serves as the base class for all Remembrance items, which are special items that can be traded for unique effects. Defines the contract for applying trade effects through the applyTradeEffect method.	**Alternate Solution**: Implement unique effects directly within each Remembrance item class. **Finalised Solution**: Create a Remembrance abstract class to define a common interface for applying trade effects. **Reasons for Decision**: **LSP**: Allows polymorphic treatment of Remembrance items, simplifying code that uses these items. **ISP**: Reduces dependencies by allowing each Remembrance to implement its own effect logic. **Limitations and Tradeoffs**: Introduces additional complexity in managing multiple Remembrance subclasses.
RememberanceOfDancingLion	Represents a specific Remembrance item that grants the player the Divine Beast Head weapon and increases health and mana when traded.	**Alternate Solution**: Apply the effect of receiving the "Divine Beast Head" and health/mana increase directly within the SuspiciousTrader class. **Finalised Solution**: Create a "RemembranceOfDancingLion" class that encapsulates the unique trade effect. **Reasons for Decision**: **SRP**: Encapsulates the specific trade effect within its own class, simplifying management and potential extensions. **OCP**: Allows for the addition of new Remembrances with unique effects without modifying the SuspiciousTrader class. **Limitations and Tradeoffs**: Introduces an additional class, increasing the complexity of the class hierarchy.
RemembranceOfFurnaceGolem	Represents another Remembrance item that provides the Furnace Engine weapon and a healing effect when traded.	**Alternate Solution**: Handle the effect of receiving the "Furnace Engine" and the healing effect directly within the SuspiciousTrader class. **Finalised Solution**: Create a "RemembranceOfFurnaceGolem" class to encapsulate the unique trade effect. **Reasons for Decision**:

		SRP: Encapsulates the specific trade effect within its own class, simplifying management and potential extensions. **OCP**: Allows for the introduction of new Remembrances with unique effects without modifying the SuspiciousTrader class. **Limitations and Tradeoffs**: Adds to the class hierarchy, potentially increasing complexity in managing multiple Remembrance subclasses.
DivineBeastHead	Represents the Divine Beast Head weapon, granting the player access to divine powers and allowing them to switch between different powers during combat.	**Alternate Solution**: Implement the divine power switching and special attacks as simple methods within the Actor class. **Finalised Solution**: Create a "DivineBeastHead" class that models the unique weapon behaviour and effects. **Reasons for Decision:** **SRP**: Encapsulates the weapon's logic, making it easier to manage and extend. **LSP**: Facilitates the addition of new weapons or modifications to existing ones without impacting other game mechanics. **Limitations and Tradeoffs**: Increases the number of weapon-specific classes, potentially complicating weapon management and extension.
FurnaceEngine	Models the Furnace Engine weapon, which has a chance to trigger an explosion and burn the surrounding area upon use.	**Alternate Solution**: Implement the explosion and burning effects as simple methods within the Actor class. **Finalised Solution**:Create a "FurnaceEngine" class that models the unique attack behaviour and effects. **Reasons for Decision**: **SRP**: Encapsulates the weapon's logic, making it easier to manage and extend. **OCP**: Facilitates the addition of new weapons or modifications to existing ones without impacting other game mechanics. **Limitations and Tradeoffs**: Increases the number of weapon-specific classes, potentially complicating weapon management and extension.

REQ3

UML



** *new classes are highlighted in Light Green, Modified Existing classes are highlighted in Light Blue*

REQ 3 Rationale

Classes Modified/ Create	Roles and Responsibilities	Rationale
Modified Classes - WeaponItem - WeaponArtsEnum	The role of the **WeaponItem** class is to simply act as a template for instances of a Weapon, it is an abstract class. It ensures that child classes have the ***appropriate fields***, and ***methods*** to pick the weapon up, attack, etc. The role of **WeaponArtsEnum** class is to simply hold Enum Attributes that determine if a ***WeaponItem*** has a ***WeaponArt*** associated with it.	**Changes Made** - WeaponItem now has a StateManager as an attribute - WeaponArtsEnum now has MEMENTO as an attribute **Alternate Solution** Have Memento class handle all logic of controlling the states of the wielder **Finalised Solution** Each WeaponItem will have its own StateManager, allowing each WeaponItem to uniquely control states, and is not shared across other WeaponItems. WeaponItems now can also be infused with a Memento WeaponArt.

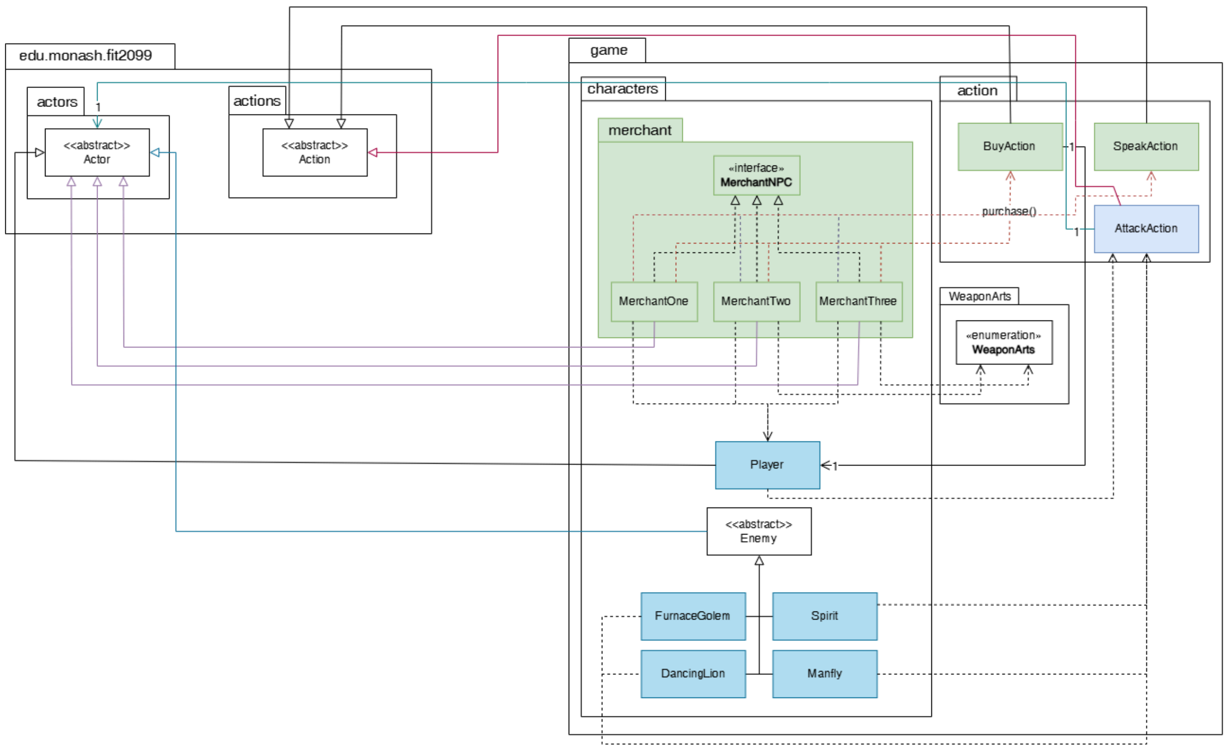
		Reasons for Decision **SRP**: Takes responsibility away from Memento class, allowing Memento to only need to handle effects on Entities, and not Weapons **DIP:** Layer of abstraction is created to prevent the need for high-level module, such as Memento to directly communicate with WeaponItem
Modified Classes - GravesitePlains	The role of **GravesitePlains** is to act as an instance of a ***GameMap***, and represents the Gravesite Plains Map. It stores its own Grounds, List of Strings to define the Map Layout, as well as a method to initialise the entities of the map, such as Actors, Items, etc.	**Changes Made** - WeaponItem's with the Memento WeaponArt are added to the Map under method initialiseEntities() **Reasons for Decision** Users of the program can now pick up and use WeaponItem's with Memento WeaponArt. No **fundamental or major changes** regarding design to this class are made, simply the addition of the weapons infused with Memento.
Created Class - Memento	**Memento** is a child class of the WeaponArt class, and can be associated with a WeaponItem. It allows weapons to store states of the game, such as the wielder, the location, and other attributes.	**Alternative Solution** Memento class controls all logic and states of the game, such as Actor Health, Location, etc. **Finalised Solution** Split up the responsibility into three different classes **Memento, StateManager** and **State** **Reasons for Decision** **SRP:** Allows the Memento WeaponArt Class to solely focus on handling what to do with a **State** of the game, along with it's properties of hurting the wielder. **OCP:** Properties from **State** class can be easily retrieved and manipulated through Memento, allowing for a more flexible design of what Memento can do in the future **DRY:** As Memento is a child class of WeaponArt, it reuses most of the existing methods provided.

Created Class - StateManager	**StateManager** manages **State** objects of the game. It holds a HashMap, utilising Actor and a List of State's as key-value pairs. It's main responsibility is to associate States to the correct Actor, as well as provides functionality to **save** and **restore** states of the game.	**Alternative Solution** Memento class controls all logic and states of the game, such as Actor Health, Location, etc. **Finalised Solution** Split up the responsibility into three different classes **StateManager, State,** and **Memento.** **Reasons for Decision** **SRP:** Has a sole purpose of managing the states of the game, and associating it with the correct Actor, taking the responsibility of other classes needing to manipulate or manage States. **OCP:** Properties from **State** class can be easily retrieved through the **StateManager**, such as in Memento. This allows for a more open, extensible design for future use in the game. **DIP: StateManager** acts as a layer of abstraction between classes that interact with **State** objects, following the idea of implementing a medium of interaction between high-level and low-level classes. Takes advantage of **Dependency Injection**, such as passing a StateManager as a parameter to a WeaponItem with Memento **WeaponArt**, allowing each individual weapon to control and manage it's own states, not having to share it.
Created Class - State	A **State** class is a class that mainly holds attributes of Actors at a particular turn of the game. This is not just limited to actors, but also the Ground, Location, and more. It works in tandem with **StateManager** to appropriately restore the states of the game to the correct Actors. It has methods to acquire the stored attributes and fields within the State object, such as getGround(), getMana, etc.	**Alternative Solution** Memento class controls all logic and states of the game, such as Actor Health, Location, etc. **Finalised Solution** Split up the responsibility into three different classes **State, StateManager** and **Memento**

		Reasons for Decision **SRP:** Has a sole purpose of storing the saved state of a turn, such as an Actor's health, mana, their location, etc. This takes the responsibility off other classes, such as Memento, or even StateManager to adhere to the SRP principles. **OCP:** Allows for an open and more extensible design with other classes, such as Memento. If it in the future would want to also manipulate other aspects of the game such as the StatusEffect of an Actor, or returning the Ground to it's original state.

REQ4

UML



*** new classes are highlighted in Light Green, Modified Existing classes are highlighted in Light Blue*

REQ 4 Rationale

Classes Modified/ Create	Roles and Responsibilities	Rationale
Created Interface: - MerchantNPC	The role of this interface is to provide and enforce a base abstraction for all the merchants when implementing them. MerchantNPC interface class ensures that all merchant class implement its methods which allows the BuyAction and SpeakAction logic to be carried out correctly. This class adheres to: **Single Responsibility Principle:** The main role of this interface is just	**Design 1:** Create an abstract class that inherits the Actor abstract class and make all the methods required by the merchant child classes into an abstract class. This makes the whole design more SRP oriented. *Pros:* - This design reduces DRY for all the child classes - Parent class handles all the logic and initialisation of inventory encouraging loose coupling

	to provide and enforce an abstraction for the merchant child classes. **Open Closed Principle:** This interface allows each individual child class to implement their own logic to use BuyAction and SpeakAction achieving OCP. **Dependency Inversion Principle:** Loose coupling of implementing this interface also makes it so the details of each merchant child class is handled by themselves.	<u>*Cons:*</u> - Double abstraction can be overly excessive causing complexity issues - Unnecessary encapsulation of child classes variables - Implementation and scalability can be a problem for more complicated logic **Design 2 (Final Design):** Create an interface so each merchant child class will implement the interface and inherit the Actor abstract class. Adheres to OCP and DIP. <u>*Pros:*</u> - Improves scalability and ease of implementation for details in each classes - Logic and readability for each child class is vastly improved allowing easier debug - Merchant each can have specification that is unique to them whilst remaining as a MerchantNPC subclass - Improve scalability as each class can have unique logic whilst adhering to this interface <u>*Cons:*</u> - If there are similarity in implementation of logic for each class there could be excessive repetition of same code - Child class must be consistent in implementation of the logic as every component of the interface is required to be implemented correctly for the child class to work
Created Class: - MerchantOne - MerchantTwo - MerchantThree	These are the child-classes that implement the MerchantNPC interface. Each of the child classes handles initialisation of their own unique inventory and dialogue, each child class is also placed on different maps. Child classes makes BuyAction and SpeakAction calls when a player interacts with it, providing it with a	**Design 1:** Make just one merchant child class that implements the MerchantNPC interface, this singular merchant child class will then handle initialisation of multiple different inventory, depending on the location this merchant is being placed on it will return a different inventory.

	menu of items the player can purchase <u>This class adhere to:</u> **Single Responsibility Principle:** Each child class is responsible for handling the initialisation of their own inventory and merchant name **Liskov Substitution Principle:** Merchant child class will return a different dialogue and list of item to purchase for the player whilst using overriding the default action list **Dependency Inversion Principle:** As each child class is their own individual class and the misimplementation of one will not affect the others due to the loose coupling from the interface	*<u>Pros:</u>* - As all the logic is in one class this makes debugging and readability a lot easier - Adhere to DRY there wouldn't need multiple MerchantNPC as one NPC can be instantiated multiple times and placed differently *<u>Cons:</u>* - Tight coupling makes scalability difficult if required - Memory overhead problem might occur when there are multiple inventory - Merchant can prompt the wrong inventory or dialogue if not implemented correctly - Implementation of this design causes a lot of problems related to the details of each individual merchant **Design 2 (Final):** Make multiple merchant child classes that implement MerchantNPC to encourage loose coupling to adhere to LSP and DIP. *<u>Pros:</u>* - Each child class can handle their own details such as individual unique inventory - Readability and scalability of this approach makes adding more vendor later easier - SpeakAction and BuyAction can be carry out differently depending on the vendor - More memory efficient as the vendor will only prompt their specific menu without going through a checking logic unlike design 1 *<u>Cons:</u>* - Can be hard to debug when there are error that occurs on specific vendors - Too loose of a coupling can cause the child class to create logical error in BuyAction and SpeakAction
Created Class:	An action child-class that handles	**Design 1:**

- BuyAction	the purchasing logic of items from merchants. This class also handles assigning of item prices and item name depending on item type by using multiple different constructor to achieve type connascence in the purchase menu when prompted The BuyAction is called in each MerchantNPC child class. It returns the items that can be purchased and the description of the item, this class calls the purchase() method in each MerchantNPC child class to handle the addition and removal of item from merchant to player's inventory. This class adheres to: **Single Responsibility Principle:** The main purpose of BuyAction is just to handle the initialisation of item prices and description, the main logic to add and remove items is handled in each individual child class. **Liskov Substitution Principle:** BuyAction class allows each MerchantNPC child class to achieve LSP as each item is initialised differently depending on their item type (type cannasense).	Handle the addition and removal item in the BuyAction class instead of making the child class handle this logic. Instead of having a purchase() method in each child class, the BuyAction() will have a purchase() method that handles updating each merchant class and player's inventory. *Pros:* - This adheres heavily to SRP as BuyAction will handle updating each merchant's inventory - Implementation of merchant npc is now really easy as they are not required to implement their own purchase logic *Cons:* - Scalability will be an issue if there are specific conditions for different merchants to purchase items from the implementation can be difficult. - Tight coupling issue can arise as BuyAction is required to know which merchant's inventory the item is from (does not adhere to LSP) **Design 2 (Final):** BuyAction is initialised for each item and BuyAction calls the purchase() method that is implemented in each individual merchant class. The BuyAction will handle the initialisation of each price and the main purchasing and dialogue logic will be called from their respective merchant class. *Pros:* - Loose coupling achieved as the BuyAction is not required to know the whole inventory of the merchant - Scalability is also a lot more better as each merchant can have their own unique purchase() logic implemented without relying on one uniformed purchase() logic - No unnecessary encapsulation required on each merchant class

		improving readability *Cons:* - Not so much on DRY as merchant classes are repeating the same logic amongst themselves - BuyAction uses an excessive amount of type connascence and can cause scalability issues if there are multiple types of items (special weapon items that are consumable etc.)
Created Class: - SpeakAction	A small action class that handles prompting an out of sale message or greeting message depending on the status of the merchant inventory. If the merchant is out of sale, the player can still speak to the merchant for the out of sale() message, no menu will be prompt and only SpeakAction will be called when the merchant is out of sale. This class adheres to: **Single Responsibility Principle:** SpeakAction class checks the current merchant's inventory status using .isEmpty() if it returns True it will prompt the outofsale() message of the merchant, else it prompts the greet() message instead. **Open Close Principle:** SpeakAction calls each merchant classes method for their own unique message allowing modifications to be made within its own merchant class whilst maintaining the same format of execution	**Design 1 (Final):** A SpeakAction action class that will automatically prompt a greeting message to the player when approached. When out of item, no menu will be prompt and only this SpeakAction will be called giving the player an option to speak to the merchant. *Pros:* - SpeakAction improves readability by breaking down the dialogue prompting logic into smaller parts - Scalability for more dialogue can be easily implemented in SpeakAction - Debugging when a dialogue is presented incorrectly is easily to do as the logic of which dialogue to be prompted is handled in SpeakAction *Cons:* - Implementation of more complicated logic that involves different dialogue require all the child class to implement that dialogue in themselves - Abstraction causes tight coupling between SpeakAction and each MerchantNPC child class this can force unnecessary implementation of useless method if a merchant does not require SpeakAction
Modified Class: - FurnaceGolem - DancingLion - Spirit - ManFly	The main modifications made to these classes is adding a balance to each of the enemies in their constructor. This balance is the gold they'll drop once defeated by the	**Rationale:** This makes addition of gold to the player once they defeat an enemy really easy, as all it requires is just to add the enemy's balance onto

- Player	player. If an enemy have no balance initialised it'll just be 0 by default.	the player's balance. Main reason for giving an enemy who dropped gold a balance is so there wouldn't be a requirement to create a specific class that handles addition, removal of gold from enemy to player.
Modified Class: - AttackAction	The main modifications made to this class is during the execute() method, where the status of the enemy is checked using isConscious(). When isConscious() returns True, the player will have the enemy's gold added to their balance.	**Rationale:** This ensures that the correct amount of gold is added to the player's inventory depending on the enemy that was defeated. This also adheres to DRY principle since no duplication or multiple IF statement required for the correct amount of gold to be added.