Requirement 4

This UML class diagram will showcase the game system in the types of classes in the game. The goal of the class system is to create a system that allows the player to choose a combat class that has different weapons and skills while allowing the possibility of extending it and be easier to maintain.

First of all the player is an actor in the game. Therefore the Player class will extend the Actor class to inherit all its properties and avoid repetitions. Each player can also choose between 3 different classes, so 3 concrete classes called Samurai, Bandit and Wretch are created that will extend the Player class since they are also players. This design will follow the Single Responsible Principle as each class (Player class) is only responsible for one type of class. For example, the Bandit class handles everything that relates to Bandit while Samurai class handles everything that relates to Samurai rather than grouping them up together into a single class. The Player class is an abstract class since it is logical that we do not instantiate a player but either a Samurai, a Bandit or a Wretch player. By doing so, the game may extend the type of classes that a player can be with ease due to the abstraction. Alternatively, we could make the Player as a concrete class while having an association with the Class (referring to the combat classes) class. However, this design is pointless as a Samurai is also a player. Moreover, to ensure the consistency in the name of the class, the enum class ClassType is there to ensure it. Alternatively, we can just use a String to represent the class type but to prevent runtime error, an enum class is recommended. Moreover, the Player class has a dependency to the Menu class since they are able to use the menu to choose their classes.

In the game, each class has their own unique signature weapon. Therefore I have created 3 concrete classes to handle the weapons attributes and methods called Uchigatana, GreatKnife and Club class that extends from WeaponItem abstract class since all weapons are a weapon item so they share common attributes. Furthermore, since a weapon is also an item, the WeaponItem abstract class also extends from the Item abstract class. Additionally, each weapon may or may not have a skill which is a capability, so it has an association with the CapabilitySet class. Alternatively, we can create a new class called skill action that each weapon has to provide the player the ability to do a skill if the conditions have been met. However, by creating a skill action which relates closely to the attack action and Capability Set this will violate the Do Not Repeat Yourself Principle. Furthermore, since a player can choose to use their fist to fight, it has a dependency with the Intrinsic Weapon class which extends the Weapon interface since a fist is also a weapon. Alternatively, we can associate a player with the Intrinsic Weapon class since dependency is a weak relationship and any entity has an intrinsic weapon.

The Player class is associated with the WeaponItem class instead of the 3 weapon class to reduce coupling. This is due to the fact of polymorphism which allows us to provide different implementation according to the type of object without the usage of if-else statement which is not recommended since it will make the Player class and the 3 weapons class to be tightly coupled. Additionally, if there are additional weapons we do not need to fix the Player class to accommodate the new weapons Therefore, we can obey the Dependency Inversion Principle and the Open-Closed Principle since the classes are hidden behind abstraction. Less effort is also required when modifying the system since the classes are independent from each other.

This UML diagram provides modularity, extensibility and reusability through polymorphism and inheritance. For example, rather than creating a class that is responsible for all of the classes, we created 3 different classes that are responsible for each class. This allows modularity as each class handles a single responsibility that can be used together to perform a larger application. This will also reduce duplicated code as there will be common attributes and be easier to maintain and test as we just only have to focus on a single module of a larger application. By using the parent class as the type, we can also allow extensibility through polymorphism by allowing the addition of new subclasses without modification on the code since polymorphism allows a code to perform a single action in different ways. Consequently, the design will adhere to the Open-Closed Principle, Single Responsibility Principle, Dependency Inversion Principle and the Do Not Repeat Yourself Principle.