Requirement 4 - Classes and weapons

Our diagram for requirement 4 is fairly simple. The requirement states that the player can select one of three combat classes for the game, each having their own weapon which can include a skill.

Combat Classes - Samurai, Bandit, and Wretch

Firstly, we created 3 combat classes, each being a child class of the abstract class playerClass. Initially, we thought it was more intuitive to make the Samurai, Bandit and so on extend the abstract Player class, but realised that this was unnecessary. The combat classes only determined the player's HP and weapon, and did not, for example, override any of the methods or add any new functionality compared to the original player class. We coupled the additional skills to the WeaponItems rather than the PlayerClasses, which makes sense as another Actor wielding the same weapon should be able to perform the same skill.

By using composition rather than inheritance, the code becomes easier to maintain and extend in the future. Creating more combat classes in the future would simply require determining the HP and weapon, rather than implementing every method from the player class, which is in line with DRY (don't repeat yourself). PlayerClass itself being an abstract class also follows the open and closed principle (OCP). Since the concrete combat classes extend PlayerClass, it is very easy to add new functionality or methods by extending the PlayerClass, while itself is abstract and thus closed for modification.

Update: This has been renamed to CombatClass, but otherwise remains the same. Our original design still promotes maintainability and extensibility by using composition over inheritance. Player class is not as heavily coupled with the weapon and HP value in a CombatClass and so this could be potentially reused on other actors in the future for additional extendability, such as allies.

Weapons - Uchigatana, Great Knife, and Club

These are all weapons which correspond to the respective combat classes, which means they extend the WeaponItem class from the game engine. Among these, 2 weapons also allow the user to perform a special skill, so a new interface Skilled was created. The Skilled interface is not exclusive to weapons, it can be implemented by anything that essentially provides the user with a new SkillAction. The weapons themselves, which extend WeaponItem, follow the Liskov Substitution principle (LSP), as all child classes of WeaponItem can act as a WeaponItem or even just an Item. In the game, this would be relevant for actions such as dropping and picking up; Uchigatana and other WeaponItems can still behave like Items for this purpose.

Update: Skills are now an optional part of weapons, and the interface is not needed. We can simply check if the getSkill() method returns null to identify whether weapons have skills or not. A possible con is that we are heavily coupling skills with weapons, but this is a good tradeoff to reduce the complexity of our design so we do not require unnecessary classes. Weapons are unlikely to share the same skill, so this form of extensibility is not as important.

Skills - QuickStep and Unsheathe

When WeaponItems implement Skilled, it means they provide the player with a specific skill. By using an interface we can easily distinguish between weapons with skills and weapons without them, and we will require them to return a SkillAction. This is another way we follow the Interface Segregation Principle as only the Weapons that return skills will implement this. Club, for example, does not provide any skills to the Player, thus it does not implement the interface and no unnecessary methods are needed.

Update: Weapons without skills are fine, as they already have the Weapon interface which, by default, returns null in the getSkill() method. Thus, weapons with skills need to override the method in order to return their skill, but weapons without skills can work in exactly the same way. This is another way to reduce complexity and improve the extendability of our code - adding weapons is very easy by not having to implement an unnecessary method if they don't have a skill, and on the other hand, adding a weapon with a skill is exactly the same by overriding the null method.

Pros and Cons

A possible con of this design is that it may seem excessive in terms of the additional classes and interfaces created, which makes the design more complex and requires more time and effort to code initially. It may be difficult to explain how all the pieces work together to a new developer. However, the flexibility and extendibility of the game are greatly improved, and this design provides a solid foundation to add more features in the future, such as additional PlayerClasses with their respective WeaponItems, Skills, and so on. Additionally, we were able to greatly reduce coupling by making use of principles like the Open Closed principle. Overall, we found a good balance between adhering to SOLID principles to gain its benefits while still considering the specific needs and limitations that were described by the assignment.

Update: This design is made simpler but also more maintainable because of the abstract CombatClass which gives any player a combination of weapon/HP values. This allows other actors to make use of CombatClasses, and the exact value can be changed easily to apply across all actors. Because of the Single Responsibility Principle and the Open Closed principle, we are able to reduce coupling by not having players depend on weapons or inheriting from Player unnecessarily.