Kevin Lew

# Recommendations to Game Engine

## SOLID Principles

### Single Responsibility Principle

Regarding the single responsibility principle, the game engine of zombieWorld satisfies the principle throughout the system design. Where most classes have one well defined responsibility.

There have been no god classes seen in the game engine, where a class contains all the logic and it manipulates other classes that are comprised solely of data. However larger classes do exist in the game engine where classes contain more methods than others, but it still satisfies the single responsibility principle as the methods all pertain to its singular defined responsibility.

E.g. Item is a large class, but the methods are all logic elements to manipulate its own data.

Actor is a large class, but its responsibility is to create an abstraction of an NPC/player which is done by the methods it contains.

By following Single Responsibility Principle, the game engine has greatly increased cohesion and allowed developers to more easily maintain and extend the system.

However, the World class contains too much responsibilities, it must process the turn execution of every turn of the game while initializing world elements and processing the actor's individual turn actions.

To fix this would require additional classes that separates the responsibilities clearly, where a class would handle processing an actor's turn actions and another would handle the initialization of the world elements.

### Open/Closed Principle

The game engine mostly adheres to the open/closed principle, the system in game engine is designed in a way for extensions to be added but not for already made code to be modified.

This is evident from the number of interfaces in the game engine, where each interface is designed to be implemented by various components and the components later inherited by extensions created by later developers. But by changing the components in the engine will create bugs and incorrect outputs in a large range of components as the game engine contains many dependencies to each other.

E.g. Many Action subclasses are all dependent on the Item class, so if the Item class were to be modified, a shotgun surgery may have to be applied to the Action subclasses as shown in figure 1.

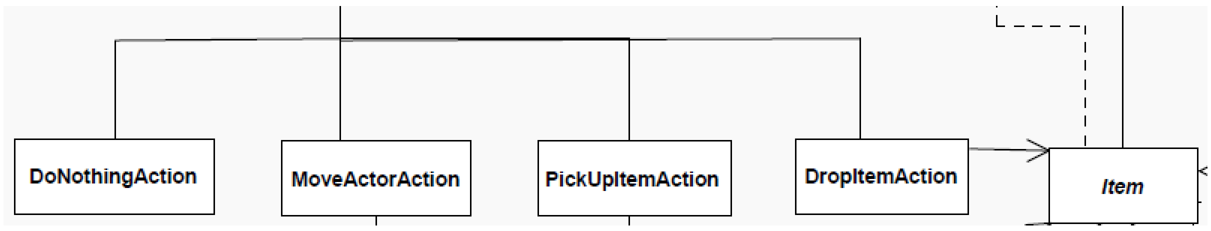


*Figure 1- Actions dependent on Item*

Although most major components such as Actor/Item are designed to be later extended with no need of modification, the world class is an offender of the Open/Closed Principle. As the World class contains a run() function that describes the execution cycle of the game but does not offer any extendibility where another class could inherit/use it to add addition elements to the cycle. So, to add new features to the game pertaining to things to execute that does not already exist in the game (e.g. actor's process turns) will require modifying the world class.

To fix this issue would be to create an interface describing the requirements for running the game and implementing the interface as a class in the game file, therefore allowing developers to extend the run execution without needing to modify the engine code.

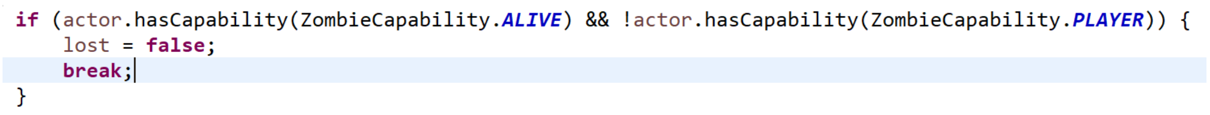
## Liskov Substitution Principle

Regarding Liskov Substitution Principle, the game engine classes that inherits are all true subtypes of their parent classes where they can implement all responsibilities held by their parent class.

Throughout the design of the game engine, there have been no classes that does not contain a method of its parent class as in Java the editor can automatically detect such inconsistencies.

Due to adhering to Liskov Substitution Principle, the engine has allowed future developers to make code easier to extend as client code can base classes when to referring to its subclasses due the subclasses being true subtypes of their parent class.

e.g. Developers can develop new Actor types but only need to know they are type Actor and not need their specific type. (For example: Zombie) As shown in figure 2, where the actor could be referencing Human, Zombie, Player or FarmerHuman.



```
if (actor.hasCapability(ZombieCapability.ALIVE) && !actor.hasCapability(ZombieCapability.PLAYER)) {
    lost = false;
    break;
}
```

*Figure 2- Base class referenced*

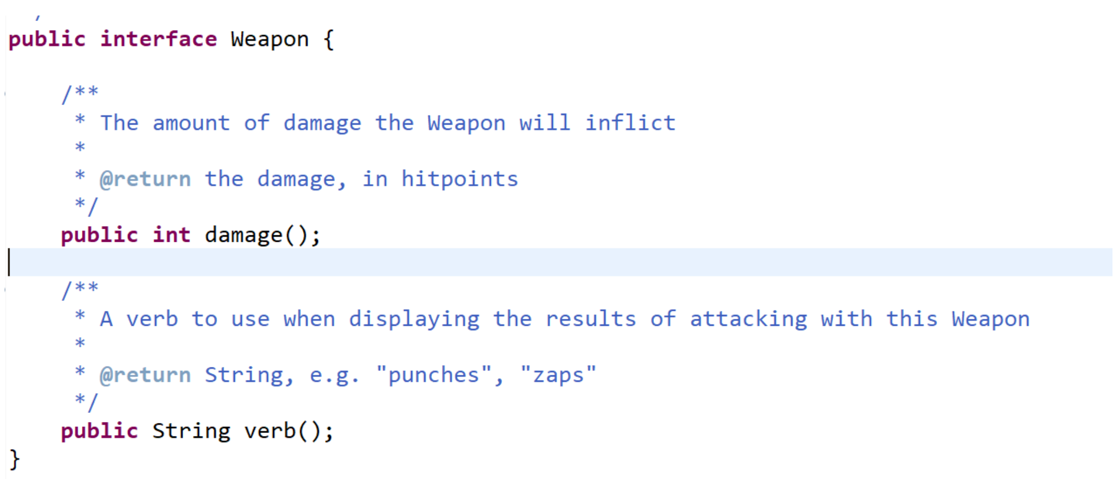
## Interface Segregation Principle

The game engine adheres to the Interface Segregation Principle as there are no excessive functions in any of the interfaces belonging to game engine, so there are no functions in an interface that is unused by the client code.

This is seen throughout the design, as the interfaces in the game engine are relatively short/small, containing only the functions that are essential to its respective abstraction. Such as: Capable, GroundFactory and Weapon. These interfaces are all concise in their functions, avoiding interface pollution.

Due to adhering to Interface Segregation Principle it's easier for developers to add new features to code without the need of refactoring the current design hence making the system more extensible.

e.g. Due to Weapon only containing StringVerb() and Damage(), developers can now add additional functions such as Shoot() in the client, but if shoot() was already in the interface, the client must implement Shoot() even if it's unnecessary as shown in figure 3.



```java
public interface Weapon {

    /**
     * The amount of damage the Weapon will inflict
     *
     * @return the damage, in hitpoints
     */
    public int damage();

    /**
     * A verb to use when displaying the results of attacking with this Weapon
     *
     * @return String, e.g. "punches", "zaps"
     */
    public String verb();
}
```

*Figure 3- Weapon Interface*

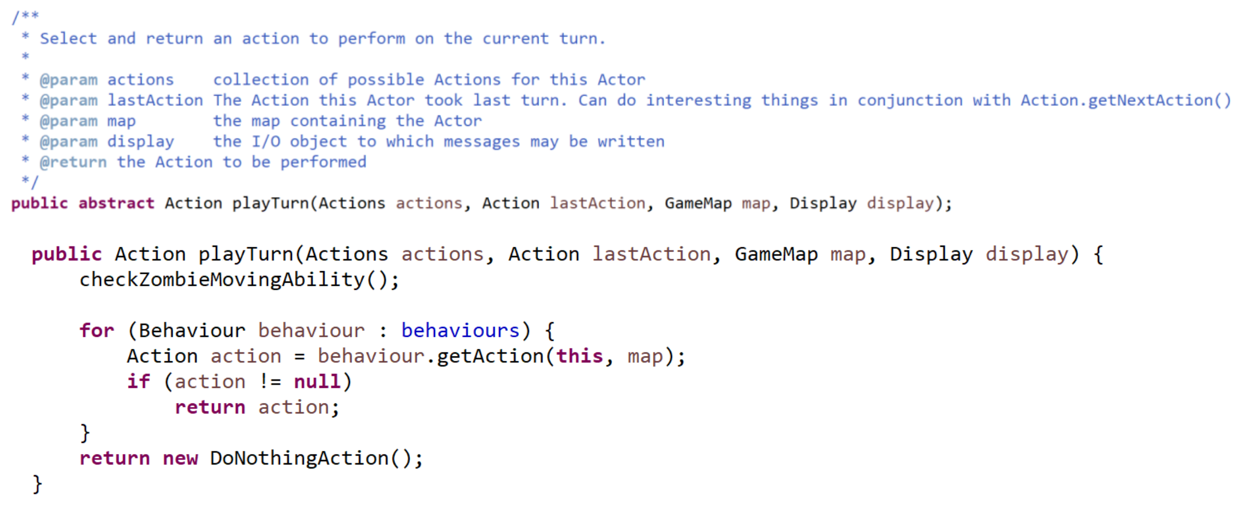
### Dependency Inversion Principle

Due to the relatively simple design of the game engine system, the dependency Inversion Principle is less applicable in this scenario. As modules in the system are simple enough that does not require to divided into high-level modules and low-level modules.

e.g. For the various Action subclasses such as MoveActorAction does not contain different types of move action, so the single class is enough to completely implement actors moving.

## Use of Abstract Classes

The use of Abstract classes is seen in the game engine and the use of abstract methods. Particularly the Abstract method playTurn() in Actor as shown in figure 1 which is later implemented by classes in the game package as shown in figure 4.



```java
/**
 * Select and return an action to perform on the current turn.
 *
 * @param actions    collection of possible Actions for this Actor
 * @param lastAction The Action this Actor took last turn. Can do interesting things in conjunction with Action.getNextAction()
 * @param map        the map containing the Actor
 * @param display    the I/O object to which messages may be written
 * @return the Action to be performed
 */
public abstract Action playTurn(Actions actions, Action lastAction, GameMap map, Display display);
```

```java
public Action playTurn(Actions actions, Action lastAction, GameMap map, Display display) {
    checkZombieMovingAbility();

    for (Behaviour behaviour : behaviours) {
        Action action = behaviour.getAction(this, map);
        if (action != null)
            return action;
    }
    return new DoNothingAction();
}
```

*Figure 4- playTurn in Actor and Zombie*

The use of playTurn() as an abstract method creates a hinge point, where the Actor class does not care about changes in the implementation that is later added in the game package, reducing likelihood of breaking Actor when later implementations are changed.

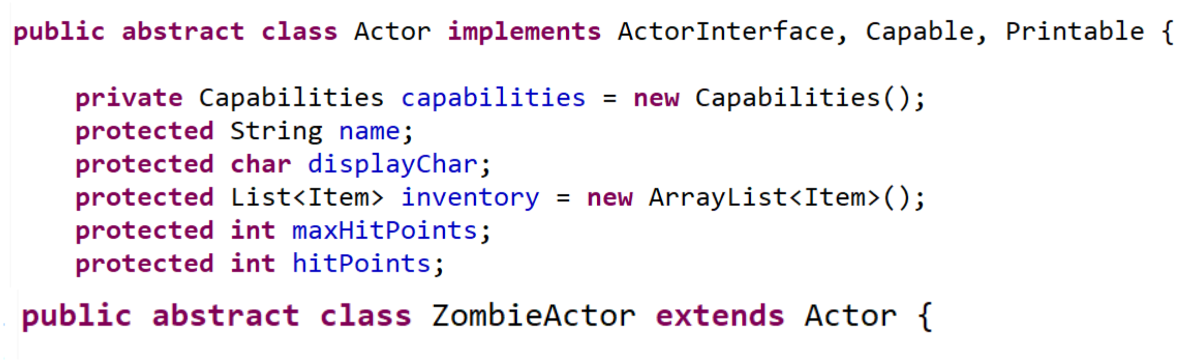
This provides the developers more extensibility as the developers can extend code and modify files in the game class without concern of the Actor class no longer providing the correct output.

## Encapsulation

The data of each class in the game engine package is well encapsulated, where the methods and variables visibility are restricted for external modules. The data is also protected by defensively copying instead of directly returning.

The variables of all classes are set to either private or protected. The private variables are well encapsulated and avoids Connascence between other classes, thus reducing coupling and increasing cohesion. However, the protected variables introduce Connascence between classes, including classes from a different package. This form of Connascence introduces severe dependence into the system and should be avoided by introducing an interface as an intermediary between the engine class and game class.

e.g. This can be seen in the Actor class, where ZombieActor directly inherits from Actor where Actor and ZombieActor are from different packages as shown in figure 5.



```
public abstract class Actor implements ActorInterface, Capable, Printable {

    private Capabilities capabilities = new Capabilities();
    protected String name;
    protected char displayChar;
    protected List<Item> inventory = new ArrayList<Item>();
    protected int maxHitPoints;
    protected int hitPoints;
```

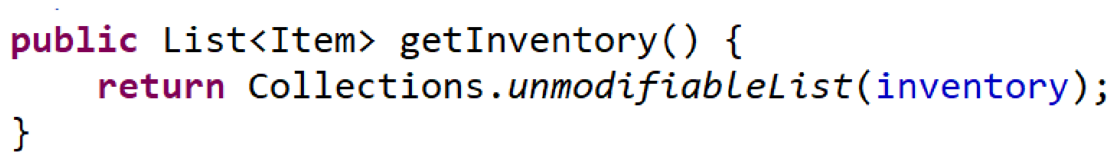
```
public abstract class ZombieActor extends Actor {
```

*Figure 5- Cross package inheritance*

The methods are mostly public, so they are not utilizing encapsulation boundaries, but although it introduces Connascence it is simple logic and not required to be modified.

Data is defensively copied instead of directly provided in the return field, avoiding any potential privacy leaks.

e.g. As seen in the Actor class in method getInventory(), where it returns Collections.unmodifiableList(inventory) instead of inventory as shown in figure 6.

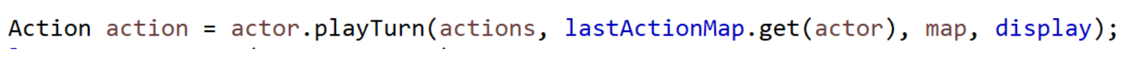


```
public List<Item> getInventory() {
    return Collections.unmodifiableList(inventory);
}
```

*Figure 6- Defensively Copying Mutable Private Object*

There are instances where the arguments may be too many.

e.g. In World playTurn has four arguments as shown in figure 7.



```
Action action = actor.playTurn(actions, lastActionMap.get(actor), map, display);
```

*Figure 7- playTurn in World*

Such cases would usually warrant the method playTurn() should be included in the World class instead of the Actor class but due to World already having multiple responsibilities, doing so would further increase World's responsibility which is unwanted. Instead some of the arguments should be initialized into Actor when constructed or through using set function and to reduce dependency the values should be done by dependency injection to remove any dependencies between World and Actor.

e.g. when Actor is instantiated, the method Actor is instantiated in can use constructor injection or later use setter injection to have the values stored in Actor instead of having World pass four arguments .