Design Rationale for Mambo Marie

Each class has their own single responsibility, for instance, `ChantBehaviour` is mainly used to generate `ChantAction`, `VoodooPriestess` to create voodoo priestess, to abide by the Single Responsibility Principle.

Liskov Substitution Principle is also enforced in `NewMap` class by inheritting `GameMap` class. Therefore `NewMap` fully behaves like `GameMap` does but with additional functionality to the map, that is to spawn `VoodooPriestess` at every turn with 5% chance. So the statement `GameMap gameMap = new NewMap(groundFactory, map)` in `Application` class can be applied.

I have also created an abstract class `VoodooPriestess` that all voodoo priestesses must implement to enforce the Open/Closed Principle, so that new functionality to Mambo Marie or any other voodoo priestesses can be added easily, but also restricting modification. Currently Mambo Marie functions mainly like a voodoo priestess, but is open for any additional modifications. (Since I was planning to add additional features such as casting spells, and leaving a poisonous creep of trail for bonus marks, but unfortunately it is closed).

Encapsulation is applied where inherritance is applied, such as `VoodooPriestess` extending from `ZombieActor`, `ChantBehaviour` extending from `Behaviour`, and `ChantAction` extending from `Action`. All attributes that are not neccesary to be made public are set to private, and getters and setters method such as `getTurns` in `VoodooPriestess` class is implemented.

Abstraction is applied to all classes, where private attributes are declared (information hiding), and getters and setters method are only declared when neccesary.

Furthermore, I also prevented privacy leak by returning a copies of the object. Example is in `ChantAction`, where i return a shallow copy of `summonedZombies` HashMap in getSummonedZombies.

```
public HashMap<Zombie, Location> getSummonedZombies() {
    // return a shallow copy of summonedZombies
    return new HashMap<Zombie, Location>(summonedZombies);
}
```

And defensive copying is implemented in constructor such as `NewWorld` so that getter methods such as `getWorld` in `NewMap` class returns only a copy of `NewWorld`.

```
public NewWorld getWorld() {
    return new NewWorld(this.world);
}
```

I have also refactored the code to apply the DRY principle in `ChantAction` class, where i seperated the task of placing zombies on the map from the `execute` method into the `placeZombies` method.