OOSE ASSIGNMENT 2017:

Name: Christopher Chang ***Student ID: 18821354*** ***Date: 12/10/17***

Introduction:
The following report is about the Combat Tool implementation. It will discuss how I used polymorphism, which design patterns, testability and plausible alternative design choices.

Polymorphism
Polymorphism has been used across the entire system. With things such as using implementation inheritance from abstract classes and also interface inheritance from classes that implement an interface. These are all forms of polymorphism that allows us to decouple the System and allow people to understand which classes are responsible to do what more easily.

I have an abstract GameCharacter class that holds all the information of each player and there are subclasses of GameCharacter that determines the type of player they are (either Player or NonPlayer). There are also abstract EffectBehaviour and TargetBehaviour classes. For Effect behaviour, there is an abstract doEffect() method that their respective subclasses calculate whether the effect does the calculation for damage or healing. For TargetBehaviour, there are abstract selectCharacter() and selectAll() methods that determine whether the ability is a single target or multi target ability. Furthermore, I created an Ability interface that has an AbilityStats() concrete implementation that is used to promote additional functionality by implementing all of its. This can enhance extra functionality because if the game decides to promote an extra ability type for a community event for a limit of time, developers can just create another AbilityStats() (with a different name) class that has its own implementation of those abstract methods and when the event is over you can just remove the concrete implementation and replace it with your original AbilityStats().

Design Patterns Used
Template Pattern:
The template pattern was used to determine whether each GameCharacter is either a Player or NonPlayer, for GameCharacter they shared all the common fields except the deciding factor of its concrete type was knowing what type to store into what team. Template pattern was also used with EffectBehaviour and TargetBehaviour because they share their own implementations of heal, damage, single and multitargets.

Factory Method Pattern:
The Factory Method pattern was used to create the Ability and GameCharacter Objects. Although each subtype of GameCharacter shares common fields, it allows the subtypes to be hidden from the user. Also this decouples the system because you are hiding any instantiation from the class it is being instantiated from therefore removing any hard coded dependencies.

Strategy Pattern:
Strategy pattern is used here with the Ability class. The Ability class has abstract method calls to its concrete implementation of AbilityStats(). The reason why it is a strategy is because during runtime, when the user selects the target ability, the ability selected is either a Single, multi, damage, heal. The idea is to have the ability called, and then let the ability decide what effect and behaviour it should do. Also, the method doEffect() inside has its own implementation of whether the ability heals or does damage.

Decorator Pattern:

Decorator Pattern is demonstrated here through the classes of Ability, AbilityStats, Target behaviour and its subclasses and EffectBehaviour and its subclasses. The idea is that Ability is an interface that has AbilityStats as a base subject and you decorate the Target and EffectBehaviours by wrapping the effect and target around the base ability. Decorator is used here for extensibility purposes if in future you are allowed to stack damage taken in abilities.

Testability
Dependency Injection within the factories is used to achieve testability. Instead of having coupled code with new instances outside the factory, you can just inject it via passing in the parameters. This will help when you try to test your code and need to mock any functionality. By doing this, it allows you to replace some of your classes without letting any other classes know about them because you haven't hardcoded any dependencies.

Next, by having your GameController acting as a central controller by instantiating your other controllers (AbilityController and CharacterController) you are able to segment your system into individual parts, so each controller can be tested in isolation. Because each controller should only be doing one thing in terms of functionality. By having your controllers instantiated as close to main can also help testability because you wouldn't have to dig too far into your code to test functionality.

Finally, by using MVC you are able to achieve testability by having separation of concerns. If you know what each class is doing, and making sure that each specific class is only responsible of doing that one thing, there is lesser of a chance that you accidentally couple classes together, which can be hard to test later on.

Alternatives

1. By using the Observer pattern for checking the whether a player dies and when a team wins. You are able to decouple the view and the controller because the Observer pattern knows when to check for an event instead of what it does.

   Pros:
   - Able to have loosely coupled designs between Objects that interact.
   - No modification needs to be done to the subject to add new observers
   - For extensibility, if you want to track other things like switching sides and experience you are able to just add Observers to the list that does that
   - Inversion of control, where the controller doesn't have control and sparks event driven programming
   - You are coding to an interface which decouples code

   Cons:
   - Java's built in class Observable forces the use of inheritance
   - The order of Observer notifications is undependable
   - A change in the Observer interface might enforce changes all over your whole program so you have to plan carefully

2. Instead of using a Decorator pattern for ability, a template pattern would be better because the only difference is whether the number for damage or heal is positive or negative. Especially in my context Ability is a model class, it would be better to have it as template whereas Decorator is more for Controllers because you are adding functionality on.

   Pros
   - Allows your program to be extensible but not modifiable. Because abilities has certain classfields that are set like the numDice and numFaces but you want to extend extra functionality to.

   Cons
   - It restricts you to only allowing to extend one thing (in java).
   - Tightly couples classes to the super class