COMP2003 Assignment Writeup

Jack McNair 18927430

Question 1: Polymorphism

The most significant uses of polymorphism in this design are the Targetable, TargetingStrategy and AbilityEffect interfaces.

Since both characters and teams can be targeted by abilities, the Character and Team classes both implement the Targetable interface. The Targetable interface contains the abstract method changeHealth(). The Character and Team classes both implement this method differently; in the case of Character, it changes the character's health, while in the case of Team, it changes the health of every member of the Team. This allows abilities to target both players and entire teams, without particularly caring which is which, reducing coupling.

Abilities make use of the Strategy pattern when implementing their targeting logic and effects. Each ability has an instance of the TargetingStrategy interface. Four classes implement this interface: TargetSingleAlly, TargetSingleEnemy, TargetMultiAlly and TargetMultiEnemy. The getTarget method, implemented by each of these classes, takes in the friendly and enemy teams and outputs a list of valid targets. Similarly, each ability has an instance of the AbilityEffect interface; the Heal and Damage classes implement this interface, and resolve the ability accordingly. Using the Strategy pattern here saves a lot of code dupication, making it unnecessary to write classes for each possible combination of targeting type and effect. It also improves extendability – in order to make new effects, one need only write new implementations of the AbilityEffect interface.

Characters make use of the Template Method pattern. Player and non-player characters operate in almost exactly the same way, with one exception – players restore 5% of their enemies' health when killed, whereas NPCs restore 10%. In order to deal with this while minimizing code duplication, the Character class has the abstract method notifyObservers. This method (also part of the class's implementation of the observer pattern) notifies all observing teams that the character has died, providing an amount of healing that varies between the PlayerCharacter and NonPlayerCharacter implementations.

Question 2: Testability

Testability has been achieved by adhering to the Dependency Injection pattern throughout the design. The keyword “new” is only used in factories and main; instead, objects' dependencies are injected during construction. This means that any object can theoretically be replaced with a mock during unit testing.

Additionally, CharacterFactory and AbilityFactory have testCharacter and testAbility fields, respectively. When these fields are set, the factory will output the test object rather than their normal output. This means that even in non-mock factories are used, they can still return mock objects.

Question 3: Alternative Designs

Using the Observer pattern to alert teams of the deaths of their own characters is probably overengineering, since a character is unlikely to have more than one friendly observer at once. A simpler and possibly more sensible alternative would be to have Characters have a field determining their own team, which they could call when they die.

Abilities having a copy of their own effects and targeting logic is a mild violation of MVC; the Ability class is part of the model, but targeting and effect logic and is something that belongs in the controller. An alternative to this would be for the ability to have chars determining their targeting and effect, which they could pass to the controller when the ability is used. However, these essentially amount to a control flags; compared to a control flag, an MVC violation is the lesser of two maintainability issues.

Abilities currently use the Strategy pattern to determine their targeting and effects. They could, however, also use the Decorator pattern, with each of the ability's effects “decorating” the ability. For this version of the combat tool, which provides only basic functionality, the Strategy pattern is more appropriate, as the Decorator pattern deals with applying multiple algorithms in sequence rather than customizing a single algorithm. The Decorator pattern could certainly be useful for future expansion, however – if applied the AbilityEffect class, it could allow for an ability to have multiple effects resolved in sequence.