Assignment Discussion

Christopher Villegas - 18359884

Polymorphism

The observer pattern was used throughout the program to update various parts of the system when a particular event occurred such as when the user interface needed to be updated to show a character's health change. Making use of this pattern allowed for the event source and observing classes to be decoupled from one another as the event source didn't need to know about what parts of the system it had to update.

The strategy pattern was employed to handle ability targeting as well as the affect they had on selected targets. Using the strategy pattern here simplified the process of using abilities since each target-selection/ability effect strategy was contained in its own class, and the combat controller did not need to know about any specific details about the ability.

Target selection involved the use of polymorphism since an ability could target either an individual character or a whole team. Using the Targetable interface, both teams and individuals could be treated by other parts of the system as identical, which simplified the design.

File reading/writing made use of the template pattern, where a generic abstract class was created which handled all the common aspects of file I/O and concrete classes for handling the character/ability specific requirements. Making use of this pattern allowed for code reuse since the common functionality of the classes could be factored out into a separate class.

Testability

Testability was made easier through the use of dependency injection and incorporating interfaces where possible.

The use of the dependency injection pattern aided with testability as it allowed for mock object to be injected in place of creating actual objects. Injecting mock objects meant the methods of these objects could be stubbed to return known values as well as verifying an object's method calls.

Testability was also made easier by having similar classes implement an interface such as the Targetable interface shared between teams and characters. Using an interface meant that a particular test case involving a Targetable object would only have to be written once rather than having separate test cases for each concrete class.

Unit testing was also simplified by allowing the character factory to return a mock object instead of a real character object. This aided with testing classes that used the character factory such as the character file reader class since the mock object could be used to verify a character object's method calls and such.

Design considerations

An alternative to the strategy pattern used to implement ability target selection/ability effect was the decorator pattern. This design was suitable since there would be one class per decoration, where each decoration would either changed who the ability could target or how the ability affected the target. The drawback to this design however, is the readability compared to strategy pattern who's flow of events could easily be seen in the combat controller, compared to the decorator pattern, where it was not immediately obvious what a decoration is referring to when it calls a super method (reference to base ability, target decoration or effect decoration?). Furthermore, abilities needed to be decorated in a specific order (target selection then ability effect, since ability effect decorations must restrict target selection to either ally or enemy), thus requiring other parts of the system, such as the ability file reader to know the implementation details for ability attributes. The use of an ability factory could be used to lessen this burden, but such a class was not necessary when using the strategy pattern.

Another alternative design for handling abilities was the template method, where an abstract ability class could be made with an abstract method for target selection and another for performing the ability. The template method seems plausible since all abilities follow the common algorithm of first determining whether to target teams of individuals, then limiting the valid targets to either enemies or allies, and then finally generating a value representing the HP effect of the ability. The problem with the template pattern however, is that two abstract methods need to be implemented. This causes a problem with code reuse since each permutation of target selection/ability effect would need to be combined into one class. For example, a heal ability could either be a single or multi-target ability and thus two classes need to be created, both of which contain the same heal implementation. This is not a major problem in the current system since there are only four possible ability permutations, but if the system is to be extensible, new ability types could be considered meaning more classes would need to be created than if the strategy pattern was implemented.