Assignment 4 Essay

Brandon Walters

u1199080

The first problem with the design and logic of the code given in the assignment is a fairly obvious one: the matter of the set() function. This function, in the documentation provided, was overridden in both the remote and observable object types, but not the user_status object type. Due to the problems associated with the diamond design of the inheritance between these types, this would lead to the problem of ambiguity regarding a set() function call from a user_status object: for example, if we had a user_status object named User, then if we called User.set(), the compiler would have no clue whether we meant to call the set() function as it is defined in the remote class definition, or the observable class definition. In this case, the code will fail at runtime, as the compiler cannot make a distinction as to which set() function to use.

The second problem I identified during my work on this assignment is the problem of the get() function's usage in the observable class's set() function. In observable's set() function definition, we need to get the object's old status value to check against the new value passed in as the argument, so as to avoid redundant work if the two statuses are the same. The problem arises when we consider the possibility that any object calling this set() function may be of either the observable or user_status type: if the object calling set() is of the observable type, then the get() function that needs to be called is the one from the status type, as an observable object will not have access to the remote type's overridden version. If the object calling set() is of the user_status type, then the get() function that needs to be called is the one from the remote type, as a user_status object needs to utilize the remote class's capabilities in order to fulfill its inheritance requirements, as laid out in the design specifications.

To fix the first problem described above, I used the principles of virtual inheritance. In the declarations of the remote and observable classes, I used the virtual keyword in front of the set function declaration, as well as in front of the status keyword in the inheritance declaration for the remote and observable classes. This means that only one copy of the underlying status object's data is created, and the remote and observable types both share the data of the one status object, instead of creating one object for each. This avoids redundant data copying, and once we also create an overwritten version of the set() function in the user_status class, we can ensure that there is no ambiguity for the compiler when we make a call to our user_status object with a set() function call.

For the second problem I faced, that of the need for the get() function call in the observable implementation of set() to be dynamic: if the object calling is a user_status object, it needs to use the remote implementation of get(), if it is just an observable object it needs to use the base status implementation. To solve this problem, I simply used the basic get() call. Unlike most of the other code in the class implementations, which would specify which class the call should use (eg: status::get()), I kept it generic. This allows for inheritance to properly operate on the get() function call and behave as the program is supposed to, which allows for the user_status functionality we expect from our inheritance, while also making sure that the observable class can work on its own and maintain expected functionality.

Overall, these problems with the initial design of the classes were solved through an application of the techniques we learned in class, such as virtual inheritance, and I was able to get the classes working together in a way that meets user expectations and matches our design intentions and goals.

Here is the updated diagram: