EECS 3311 Lab 3 Design Decisions

How is the undo/redo design pattern implemented in the model cluster?

In the model cluster the undo/redo design pattern is implemented with a pattern in which supports multi level undo and redo, being usable to a wide range of applications regardless of the application domain and doesn't require any redesign for new commands into our application. The first step to take is to find an abstraction for pattern. This is done in my model cluster with the “OPERATION” class where it is a deferred class and has deferred routines such as redo, undo and execute. Doing this will allow me to create effective classes as descendants that inherit the “OPERATION” class that are operations to be used for any undo/redo command. This will allow me to easily add and remove commands without having to change the implementation of my undo/redo. Now into actually manipulating the commands we must have a way to store the commands in a polymorphic collection in order to provide the depth we need in undo/redo to support multi levels. To do this we will create a “HISTORY” class which will hold the undoable or redoable operations and remove/iterate through them based on the undo/redo commands when called. This history list can be implemented with different structures, in my case I used an “ARRAYED_LIST”. The idea is to add the command in which you performed to the history list if eligible and the list will have a cursor in which will move based on any undo/redo command. The benefit of this is that now with the cursor we can move in between the history list without or with removing operations depending on the situation (Multi or Single Level Undo/Redo). When an operation in the history list is called then the undo, redo command will be called from the operations dynamic type. Overall this design pattern is very effective helping us satisfy design principles like modularity and single choice principle.

How are polymorphism and dynamic binding realized in your design at compile time and runtime?

In my design polymorphism and dynamic binding play a major part in how they are represented at runtime and compile time. Within my model where I used a “GRID” class to represent all of the data on the game, the features in the grid are dynamically bound at compile time as the “current shape” of the features are known to the compiler before runtime. This goes the same way to polymorphism at compile time as the grid call is already realized at that point. During runtime, different features in my “OPERATION” classes are realized from polymorphism and dynamic binding. This is from the undo and redo commands where the operation class to call the feature from is determined at runtime from the dynamic type because the compiler is unsure of which one to call. The object variable for operation which has multiple allowable dynamic types demonstrates polymorphism as many different versions of the features can be called for it. This was then used to put into a polymorphic collection in which the different types of operations could be added into and then whatever feature had to be executed would be determined at runtime from the operations dynamic type. Overall polymorphism and dynamic binding played a large role in my design and allows me to expand expectations, call different features and prevent code reuse in my code.