**NotePad Undo/Redo Design Document**

Ian Cowan

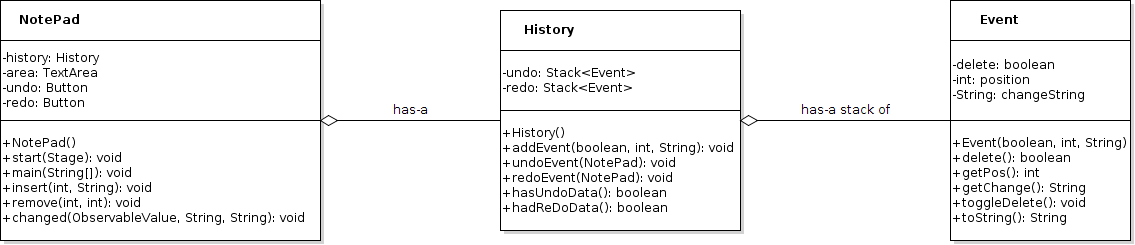
**Purpose**

A simple NotePad has been created, but it has been created without an undo or redo function and we would like to add that functionality. Anyone that uses this NotePad application will be able to use our undo and redo implementation to be able to correct mistakes in NotePad without having to remember what went wrong. This should be able to undo and redo changes in NotePad, and it should go one character at a time. Redo should allow a correction of an undo, whereas after continuing after redoing something should no longer allow redo to be used. While using redo, the user should be able to undo the most recent redo.

**Specifications**

Our History class should integrate with the already created NotePad class. The NotePad has already been written, and our History class will simply implement the undo and redo abilities. The History class should have five methods, AddEvent(), undoEvent(), redoEvent(), hasRedoData(), and hasUnDoData(). The AddEvent() method should simply add an event when called, and this event will be stored so it can be undone. This method will be called when a change is made within the NotePad. The undoEvent() method should undo the most recent event that has been created, and upon undoing the event, the event should be saved as an undo so it can be redone if a mistake was made. The redoEvent() method is very similar to the undoEvent() method, but it redoes the last event that was undone. The hasRedoData() and hasUnDoData() methods do exactly the same thing, but one with redo data and one with undo data. They both return true if data exists, and false if data does not exist. This will be called by the NotePad to check if the appropriate button should be active or not.

**Design Overview**

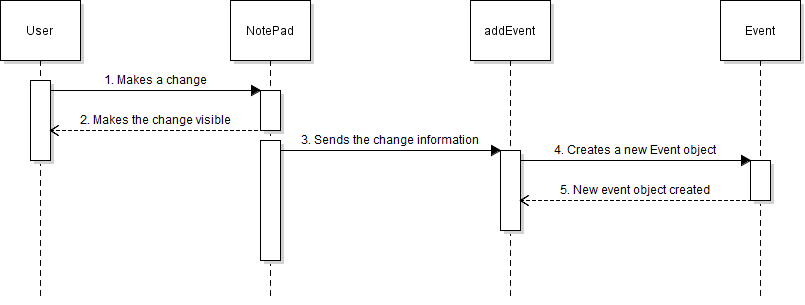


**Figure 1. Class Diagram for NotePad with History**

In order to implement an undo ability for the NotePad, I had to implement the predefined methods within the History class. The class stores two private stacks of changes the user has made (undo) and undos the user has done (redo). The objects stored inside of these stacks are events that I created with the Event class. The Event class stores all of the applicable information for the History class to be able to know what actions to carry out. Various methods within the History class call methods within the NotePad class to allow for insertions and deletions of the information from the user. Overall, this is a fairly simple but effective design.

**Detailed Design Overview**

Within the History class, we must be able to add events, undo events, redo events, and check if there are events stored to undo and redo. Within the addEvent() method, we take a boolean, int, and String value that designates whether or not the user deleted the text, the position of the change, and finally the actual change itself. Then an Event object is created that stores all of the information for the undo when the event is called. The delete attribute of the object is the opposite of the provided boolean because if the user deleted something, we want to insert it on redo and vice versa. The position and changeString attributes are exactly the same as the corresponding values the method takes.



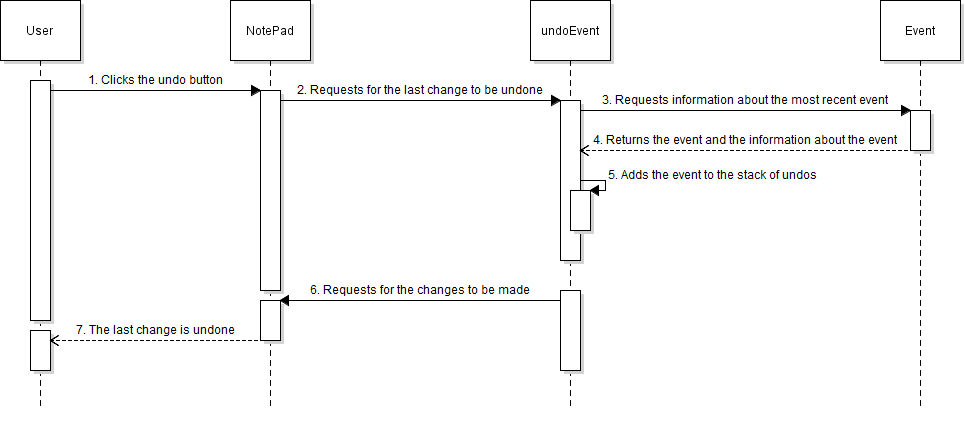
**Figure 2. addEvent() Sequence Diagram**

When the undoEvent() method is called, the decision between whether a deletion or insertion is needed has to be made. This is controlled by an if statement and the isDelete() method on the event.



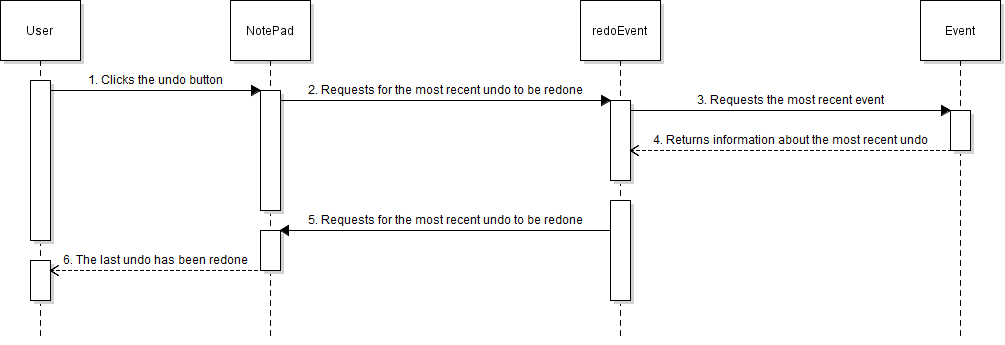
**Figure 3. Pseudocode for the if-statement within undoEvent() method**

After undoing the event, the event deletion will be toggled with the toggleDelete() method and then added to the redo stack so if the redoEvent() method is called next, it will be on the top of the stack to redo.



**Figure 4. undoEvent() Sequence Diagram**

When the redoEvent() method is called, the same process happens but with the event in the top of the stack of redo instead of undo. The event’s delete method is not toggled when redo happens, and the event is not added to the undo stack.



**Figure 5. redoEvent() Sequence Diagram**

This implementation will result in the appropriate undo and redo functionalities within the NotePad.

The last thing left is for the NotePad to be able to tell if there is anything that can be undone or redone to prevent an error from disrupting the NotePad. This is done with two simple methods, hasUndoData() and hasReDoData(). These check to see if their respective stacks are empty and if so, they return false, for no data, and if not, they return true, for data.



**Figure 6. Pseudocode for the if-statement that checks if the respective stack is empty**

**Analysis**

Most of the undo and redo functionality was fairly straightforward, although some of the things could have been done differently. The Event class could have been called something else, and the way the data is stored for events could have been done differently. The delete attribute could have been saved as the action that was taken, instead of what needs to be taken, but I made the decision to have it saved as the action that needs to be taken. I believe that this was an appropriate design choice because it allows easier debugging since the “flipping” of the value can be done anytime assigning the attribute is done.

The design choice to make a class that stores the events was done to simplify the storage of all the necessary information. The stack could have stored object arrays, but that adds the need for masking the variables when getting them, and other issues that could go along with getting the appropriate values into the arrays. Creating a new class was the simplest and most effective way to get this done without using arrays.

The time complexity of this entire project is constant time, . There are no loops within this implementation and the amount of time any individual method will take will not be affected by the inputs. If the user is working on a very long and complex project within the NotePad, the memory could fill up with all of undos and redos, but since the amount of data stored within each event takes up a fairly small amount of memory, it is very unlikely the user would run into this issue for the intended use. If this was intended to be scalable to a word processor, some work would have to go into optimizing the memory usage because if someone was typing a long paper, the memory could fill up fairly fast.

The only risk with this project would be the memory filling up and not being able to store any more events. This is highly unlikely to occur for the intended usage, but as stated above could be an issue if the project was to be scaled.

**Conclusion**

Although this project may not be scalable, there are no plans to scale it up any time soon and it meets all of the required specifications. The undo and redo functionalities integrate with the NotePad as expected and everything works properly. Since the time complexity is constant, there is not a chance that later down the road there will be an issue that occurs because it is taking too much time to carry out a process. Overall, this project was a success and all of the requirements have been met.