**Lab4 Design Document**

Xu Song

**Purpose**

The purpose of lab 4 is applying the feature of stack which is the LIFO structure on the notepad’s undo and redo buttons. The user can type words on the notepad and delete them, and by clicking corresponding buttons, they can undo or redo their previous action. The undo and redo are the basic functions that widely used on almost all the editing apps, for instance Word, Power Point, web browsers and so on. They are convenient for user to do the opposite action of the previous one or simply change the state back and forth.

**Specifications**

To fulfill the purpose, the core design is that there should have two memory storage units represents redo and undo that can store the previous actions. Whenever there is a change on the notepad, the undo memory storage unit should memorize the action. Later when user click on the undo button, the notepad will do the very last action stored in the undo memory storage unit inversely. The redo button will do the very last end action done by undo. Additionally, if there is a change on the notepad that is not done by clicking redo or undo button, the redo memory storage unit will be cleared. Here is an example. Suppose user types word “apple” on the notepad. If click undo, letter “e” will be deleted; then click redo, letter “e” will show up again. However, if the user now type a “k” after “e”, since there is no undo action take at this point, the redo button can’t be clicked. Thinking through the example, undo and redo did the same string change in opposite way and will pop the last action they do to the other. The fact that they all modify the notepad according to the very last action, the memory storage units represent these two button will fit the stack structure well.

**Design Overview**

图片包含 屏幕截图

描述已自动生成

**Figure 1. Class Diagram for lab 4**

**图片包含 文字, 地图

描述已自动生成**

**Figure 2. Sequence Diagram for lab 4**

To implement undo and redo, we modify the History class methods provided. The NotePad class has been written and can’t be edited. A private attribute of NotePad is a History object and it is only called the **addEvent()** method in the **change()** method in the NotePad. (the **change()** method won’t be called in code, so not show on class diagram). **addEvent()** method takes three parameters, Boolean deletion indicates if the action is insert or remove, int position gives the position where the change happens, and String Change gives the exactly change content. Since there are three parameters need to track for every action, I created three stacks for each parameter for redo and undo. In the **addEvent(),** whenever the change happens, three stacks of undo will push corresponding variables into stacks, and clear all memories inside three stacks of redo. For both **undoEvent()** and **redoEvent(),** a NotePad object is passed into as a parameter; in this case, only note public method **insert()** and **remove()** can be called. In **undoEvent(), t**hree stacks of redo will push what pop from three stacks of undo correspondingly. If the deletion is true, the object calls **insert()** method, and pass another two paired parameters in to the method, vise versa. In **redoEvent()**, three stacks of undo will push what pop from three stacks of redo correspondingly. In redo action, if the deletion is true, the object calls **remove()** method, and pass another two paired parameters in to the method, vise versa. The last two methods, **hasUndoData()** and hasReDoData(), simply check one of three stacks of undo and redo is empty or not. (three stacks are always change together) If retunes false, the corresponding button can’t be clicked by the user.

**Analysis**

An alternative way I did previously is create another class called Node. The Node include three attributes which are the three parameters of the **addEvent().** In this alternative method, there is no need to create three stacks for undo and redo. The History class will then only have two stack attributes each represents undo and redo. Each action will be stored as Node objects in stack. The benefit is that the code will be simplified. Each method will be almost the same, the only difference would be instead of popping and pushing elements in stacks three times in a method, now just pop and push the Node object once to the other stack as needed. The risk is that in order to get one attribute of a Node object as parameter to pass into required method, the code will be long. For example, you want to get the length of change, you need “undo.pop().change.length”, which will easily make mistakes. Although the method I pick ultimately need much more code, it’s easy to check and debug.

The time complexity for both methods are O(1), because there is no loop inside the method.

**Conclusion**

In this design document, I lay out the purpose of the project and then describe the overall layout of out UML class diagram including relationships of classes and objects and how each method is implemented and called. I also compare and contrast benefits, risks and time complexities of alternative designs. To sum up, the design can be considered as successful; it passed the unit testing and fulfills all the requirements that are asked.