NotePad Design Document

Jenna, Leah, Joe

Purpose

The purpose of this project is to extend the existing **NotePad** application by adding functionality for undoing and redoing user actions. These features will allow the user to revert changes to their text, providing greater control over their text editing experience. By introducing **Undo** and **Redo** functionality, the NotePad application will become more efficient and user-friendly. It will make the editing process more flexible for the user by letting them fix mistakes with the click of a few buttons.

Specifications

- The NotePad system has been successfully implemented. We are now tasked with adding undo and redo functionality without altering the existing NotePad class and without changing the public signatures of the methods in History.

- The NotePad application would like to allow users to **Undo** their last action, such as text insertion or deletion, with a button click. This will give users the ability to quickly revert to the previous action and states of their document.

- The NotePad application would like to also allow users to **Redo** actions that were previously undone. This can help users recover any changes they want to re-apply efficiently.

- Our algorithm must store all user actions in a way that supports both undo and redo capabilities without slowing down the application's performance.

Design Overview

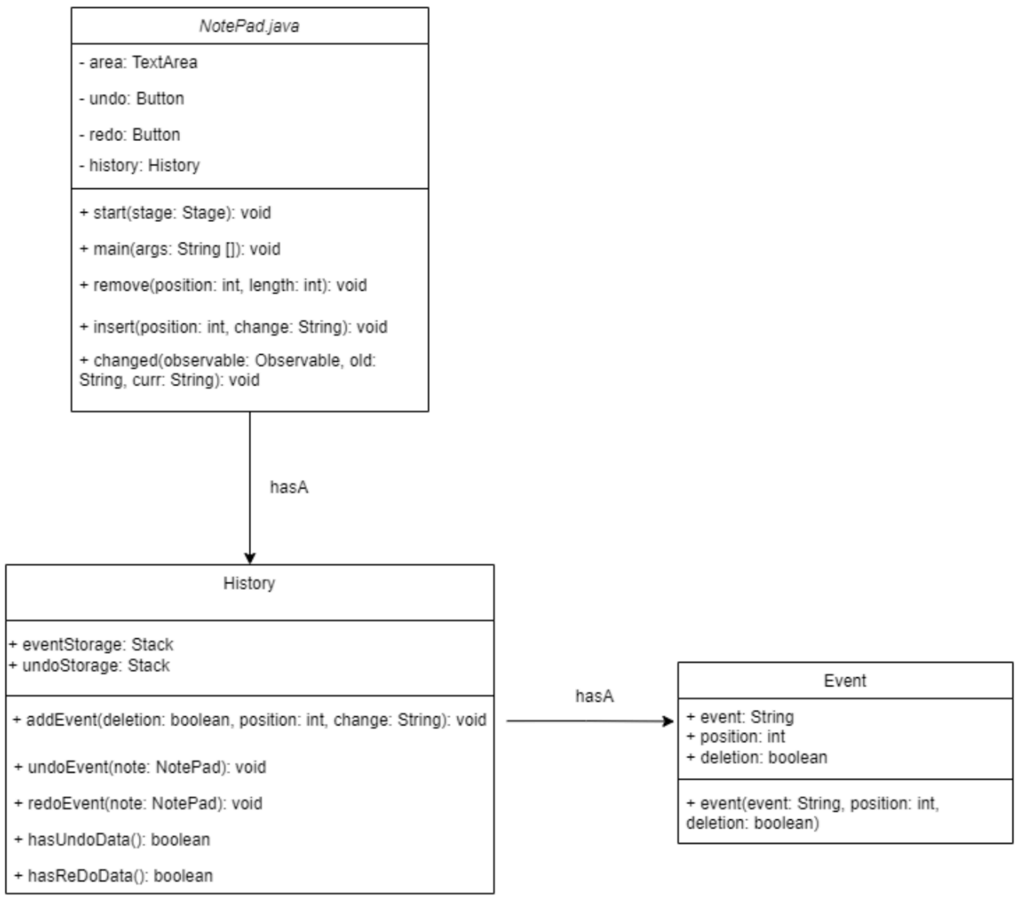
NotePad.java
- area: TextArea
- undo: Button
- redo: Button
- history: History
+ start(stage: Stage): void
+ main(args: String []): void
+ remove(position: int, length: int): void
+ insert(position: int, change: String): void
+ changed(observable: Observable, old: String, curr: String): void
hasA
History
+ eventStorage: Stack
+ undoStorage: Stack
+ addEvent(deletion: boolean, position: int, change: String): void
+ undoEvent(note: NotePad): void
+ redoEvent(note: NotePad): void
+ hasUndoData(): boolean
+ hasReDoData(): boolean
hasA
Event
+ event: String
+ position: int
+ deletion: boolean
+ event(event: String, position: int, deletion: boolean)

Figure 1: Class Diagram for Existing and Proposed Systems

Figure 1 contains the UML diagram for the existing system and the proposed additions which would enable support for Undo and Redo features of the NotePad program. The proposed system utilizes the existing system and only requires minor changes to the existing system.

Existing System

The existing system (on the left of Figure 1) consists of two classes, **NotePad** and **History**. NotePad is responsible for drawing up the NotePad window, initializing buttons, and recognizing when changes to the window have been made (e.g., the user types into the text window). NotePad also has the capability to remove and insert text.

NotePad interacts with the History class in a few cases. One case is to communicate to History when changes have been made in the window via the method `changed`. Other cases are when the “Undo” or “Redo” button are pressed in the window, at which point NotePad calls History’s methods, `undoEvent` and `redoEvent,` respectively. History also has methods for checking whether there is data stored for the Undo and Redo actions, `hasUndoData` and `hasRedoData`.

Proposed System

We propose two new fields to History, as well as an additional class (on the right of Figure 1), in order to integrate Undo and Redo functionality into the NotePad program.

History's two new fields, `eventStorage` and `undoStorage`, are Stacks that serve to store the actions of the user. The `eventStorage` Stack tracks each action a user makes within the NotePad window, be it typing a word, copy and pasting, deleting a portion of what they have written, etc. The `addEvent` method will help discriminate these actions from one another and store them in individual Stack frames properly. Meanwhile, the `undoStorage` Stack tracks each Undo action the user makes, allowing for the functionality of the Redo feature.

Event is the class for objects which are stored in the History Stacks. Each Event contains an event String, the position at which the event occurred in the NotePad text, and whether the event consisted of a deletion action.

Integration

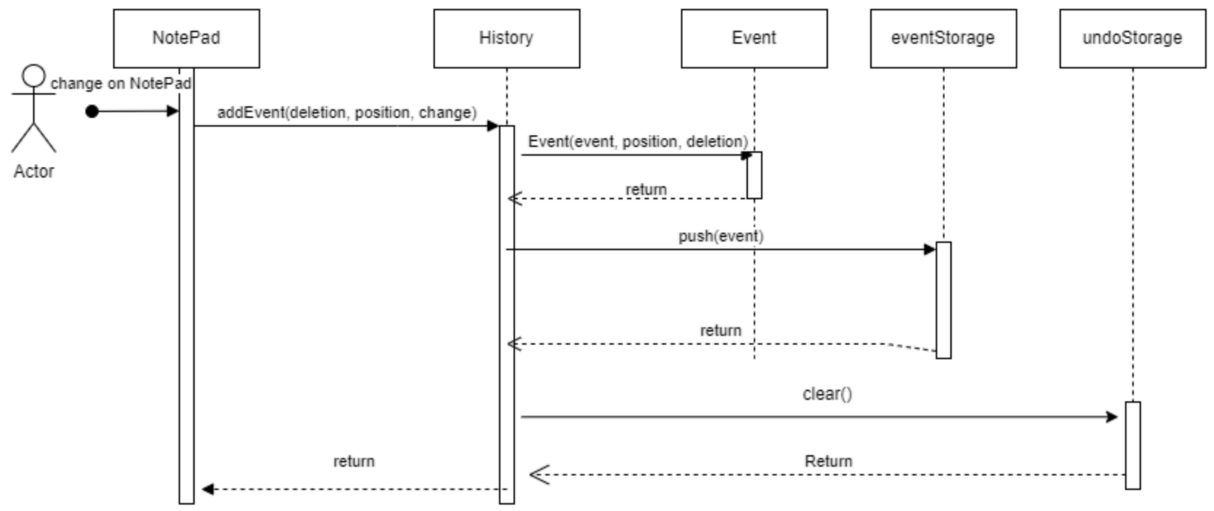

Figure 2: Sequence Diagram for addEvent in the Proposed System

Figure 2 describes how History's method, `addEvent`, will interact with the proposed system. `addEvent` will create an Event object and store it in the `eventStorage` Stack via `push`. In addition, when the user makes a new action in the window, the `undoStorage` will be `clear`ed. This process will happen repeatedly as the user continues to act within the NotePad, effectively recording each action the user makes.

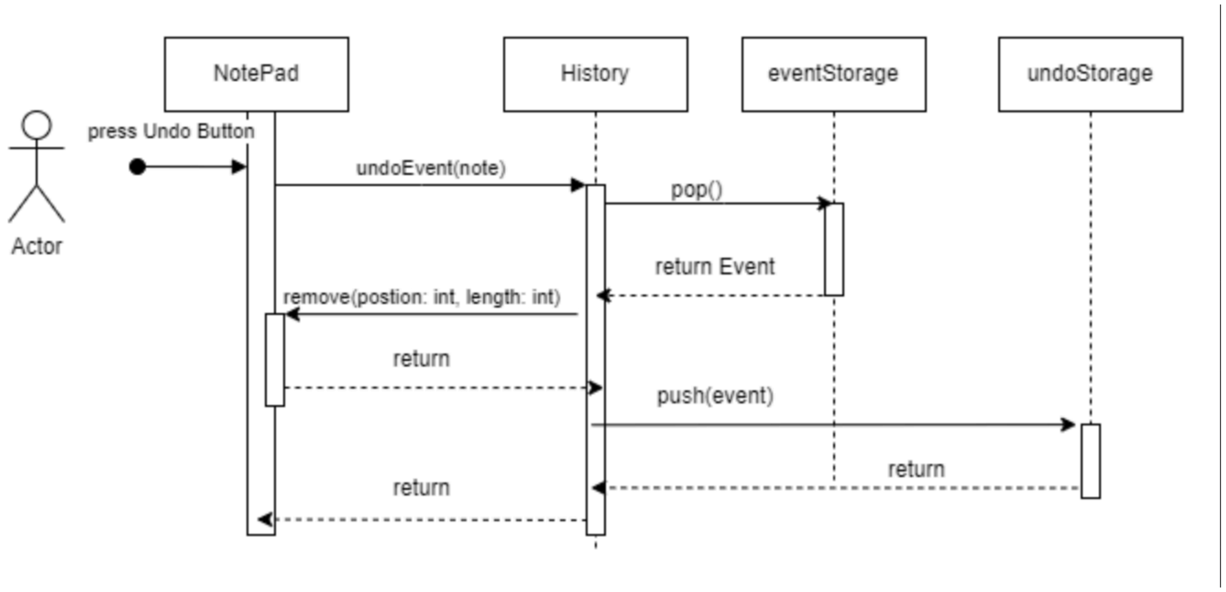

Figure 3: Sequence Diagram for undoEvent in the Proposed System

Figure 3 describes how the “Undo” button will function in the proposed system. When the “Undo” button is pressed, History’s method `undoEvent` will access the `eventStorage` Stack, `pop` the most recently stored Event, and `remove` the contents of the Event in the NotePad. In order to ensure data is not lost (in the case that the user decides to redo their undone action), the removed Event is `push`ed onto the `undoStorage` Stack. This process repeats for multiple, consecutive clicks of the “Undo” button.

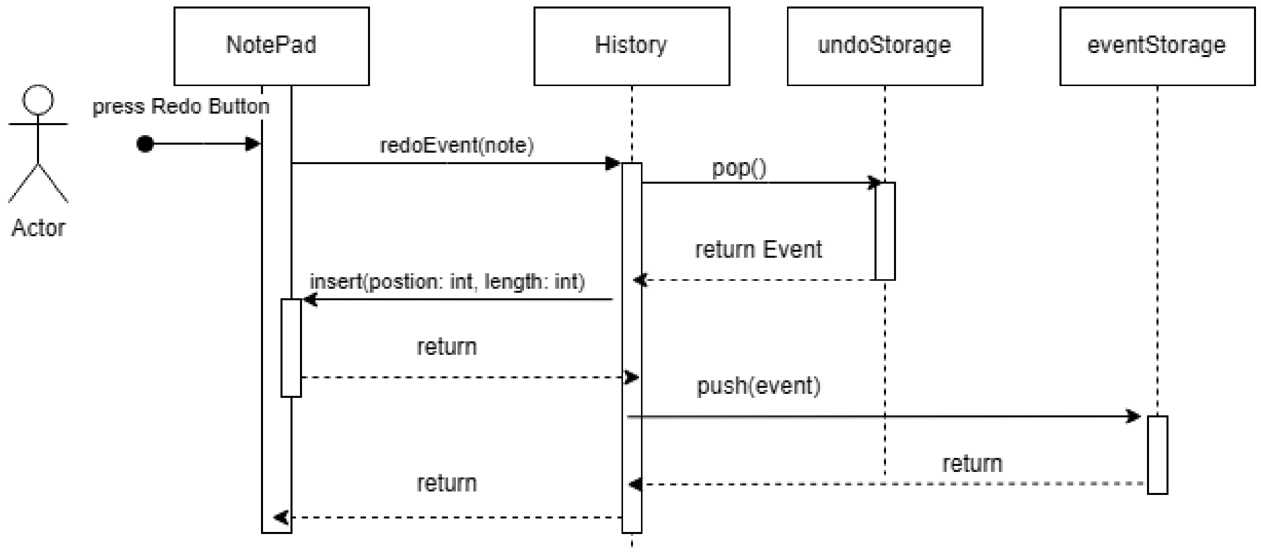

Figure 4: Sequence Diagram for redoEvent in the Proposed System

Figure 4 describes how the “Redo” button will function in the proposed system. When the “Redo” button is pressed, History’s method `redoEvent` will access the `undoStorage` Stack, `pop` the most recently stored Event, and `insert` the contents of the Event in the NotePad. In order to ensure data is not lost (in the case that the user decides to redo their undone action), the inserted Event is `push`ed onto the `eventStorage` Stack. This process repeats for multiple, consecutive clicks of the “Redo” button.

Subsystem Specifications

Since we do not need to access elements at any position other than that at the “top,” we have decided the most suitable data structure for this project is that of a Stack.

For this system, we are using the **Stack** class from the Java Collections framework to store Events. Specifically, we are using the `pop`, `push`, and `clear` methods from the Stack class. `pop` allows us to simultaneously access and remove the most recently stored Event from the Stack; `push` allows us to add an Event onto the top of the Stack; `clear` allows us to clear the entire Stack of its stored data.

Summary

The addition of a new class, Event, enables us to succinctly store information about any actions a user might make in the NotePad, such as the text, position, and whether said action was a deletion. Stacks allow for quick access, retrieval and addition of data. We have two Stacks in our proposed system which allow us to store and track both Events as they are happening and Events which have been undone so that they can be redone. The use of Stacks also allows us to easily clear out the storage of all undone Events whenever changes are detected in the NotePad. The implementation of these Stacks should allow for full redo and undo capabilities for the NotePad. These proposals are designed such that minimal changes are made to the existing system.