Undo

Interaction Techniques and Technologies (ITT), SS 2017
Session 20 (18.07.2017), Raphael Wimmer

Overview

These are slides/notes for the lecture, automatically generated from the slide set. Please extend this outline with your own notes.

Overview

- Undo
  - History
  - User Interfaces
  - Implementations
- partially based on slides by Brad Myers
  Undo: History and Models

Undo

Questions:

- What is *undo*?
- How does the user interface for undo look like?
- How can undo functionality be implemented?

History

First documented use: Bravo text editor (Xerox Alto), 1974 (user manual)

Shortcuts

- NY Times (quoted by Wikipedia) says shortcut ^Z was selected „by programmers at the research center Xerox PARC"
  - http://www.nytimes.com/2009/09/20/magazine/20FOB-onlanguage-t.html
  - Larry Tesler says that is incorrect: the Ctrl-CXVZ shortcuts were implemented first for the Apple Lisa (1983).

- Redo shortcut
  - ^Y in Macintosh
  - ^-SHIFT-Z in some other systems
- Design Issue: how big a unit to undo?
  - Often typing coalesced into a single operation
  - Multiple backspaces may or may not be
  - Newer: “intelligent” single operations may be divided into multiple undoable operations
  - E.g., Auto-correct in Word

<small>(source: Brad Myers' slide set)</small>

Linear Multi-Level Undo Model

- All operations are in a history list
- Can undo backwards
  - Undone operations are put into a redo list
- Can then redo forwards
- But once a new command is executed, anything in the redo list is discarded, so there is always only a linear history
- May have a limited size of the history list
- Almost all of today’s applications support restricted linear undo model

<small>(source: Brad Myers)</small>

Important details

- Which commands are designed to be undoable, which not?
  - mostly undoable: modify text, delete objects
  - mostly not undoable: save file, select text, send e-mail
- How are commands handled which are not undoable?
  - exclude from undo stack
- What happens to clipboard contents?
  - leave clipboard unaffected from all undo operations

Preserving the complete command history

- Problem: undo followed by other operations overwrites part of the linear undo stack
- Emacs text editor: undo operations are appended to the undo stack similar to normal operations
  - very confusing for new users
- Vim text editor: undo branches (move along main branch with u and `Ctrl-R`, move chronologically through all branches via `g-` and `g+`)
  - also offers 'persistent' undo by storing all operations in an `undofile`

Selective undo

- Let the user select which operation(s) to undo while leaving later operations intact
- „Script“ model – pretend the operation never happened
  - Can undo all operations to that point, remove the command, then redo all the subsequent commands
  - „Rewrite history“
  - But what if it was a “create” and later operations were “change color”?
    - Not allowed to selectively undo the create?
    - Or later operations are ignored?
    - Not always clear what the user would want
  - Can also support “insert” operation into history
    - If I create an object in the past, do future operations include it?
- „Inverse“ Model: add inverse operation to end of history
  - e.g., „change color from red to blue“ → „change color from blue to red“
  - see Thomas Berlage. 1994. A selective undo mechanism for graphical user interfaces based on command objects. ACM Trans. Comput.-Hum. Interact. 1, 3 (September 1994), 269-294

<small>(source: Brad Myers)</small>

Multi-user undo

- Multiple users editing at the same time
- When user A undoes something, what does it mean?
  - Local: That person’s last operation?
  - Global: Globally the last operation?
- Abowd proposes global when there is a single cursor (or single selection), but local if multiple cursors (selections)
- Local undo requires some form of selective undo
- Can interfere with the other user’s current edits
- see Gregory D. Abowd and Alan J. Dix. 1992. Giving undo attention. Interact. Comput. 4, 3 (December 1992), 317-34290021-7)

<small>(source: Brad Myers)</small>

Practical Implementations

Implementation 1: Memento Pattern

- remember each *state*
- General idea:
  - an `originator` object has some internal state
  - it can produce a `memento` object that represents its internal state
  - a `caretaker` object applies an undo-able action to the `originator` by asking for a

    `memento` object of the current state and only then applying the action
  - to undo the action, the `memento` object is given to the `originator`

* see also: (Wikipedia article)

Implementation 2: Command Pattern

- remember each *operation*
- General idea:
  - every action is encapsulated in a *command* object
  - the *command* object has specific `do()` and `undo()` methods
  - to apply an action, `do()` is called, to revert it, `undo()` is called
  - an *undo stack* organizes the sequence of commands
- similar: `diff` and `patch` utilities for code development
- see also: (Wikipedia article)
- see also: *undo branches*, e.g. in vim

Qt Implementation

- Qt's Undo framework implements the *Command* pattern.
- QUndoCommand (undo-able actions implemented as subclasses)
  - `undo()` - undoes the action
  - `redo()` - executes / redoes the action
- QUndoStack (maintains list of actions that can be undone)
  - `push(command)` - adds to stack and calls `command.redo()`
  - `undo()` - pops last command from stack and calls `command.undo()`
- QUndoGroup (route undo()/redo() to QUndoStacks for multiple opened documents)
- QUndoView (widget that shows a QUndoStack)
- also: facilities for undoing multiple related actions at once

Qt Undo Example (1/2)

```
~~~~ undo.py
#!/usr/bin/env python3
from PyQt5.QtWidgets import QUndoCommand, QUndoStack, QUndoGroup

class SimpleDocument(object):
    def __init__(self, text=None):
        if text is None:
            self.text = ""
        else:
            self.text = text

class InsertCharacter(QUndoCommand):
    def __init__(self, document, position, character):
        super().__init__()
```

```
        self.document = document
        self.character = character
        self.position = position
        self.setText("insert a character")
    def undo(self):
        self.document.text = self.document.text[:self.position] \
                           + self.document.text[self.position+1:]
    def redo(self):
        self.document.text = self.document.text[:self.position] \
                           + self.character \
                           + self.document.text[self.position:]
~~~~
```

Qt Undo Example (2/2)

```
~~~~ undo.py

if __name__ == "__main__":
    stack = QUndoStack()
    d = SimpleDocument("123456")
    stack.push(InsertCharacter(d, 1, "a"))
    # "1a23456"
    stack.push(InsertCharacter(d, 3, "b"))
    # "1a2b3456"
    stack.undo()
    stack.undo()
    # "123456"
~~~~
```

Recap

- Undo is an essential interaction technique in modern user interfaces.
- When should one use the Memento pattern, when the Command pattern?
- Qt: command pattern
- see also: C# examples for Command and Memento Pattern

ENDE