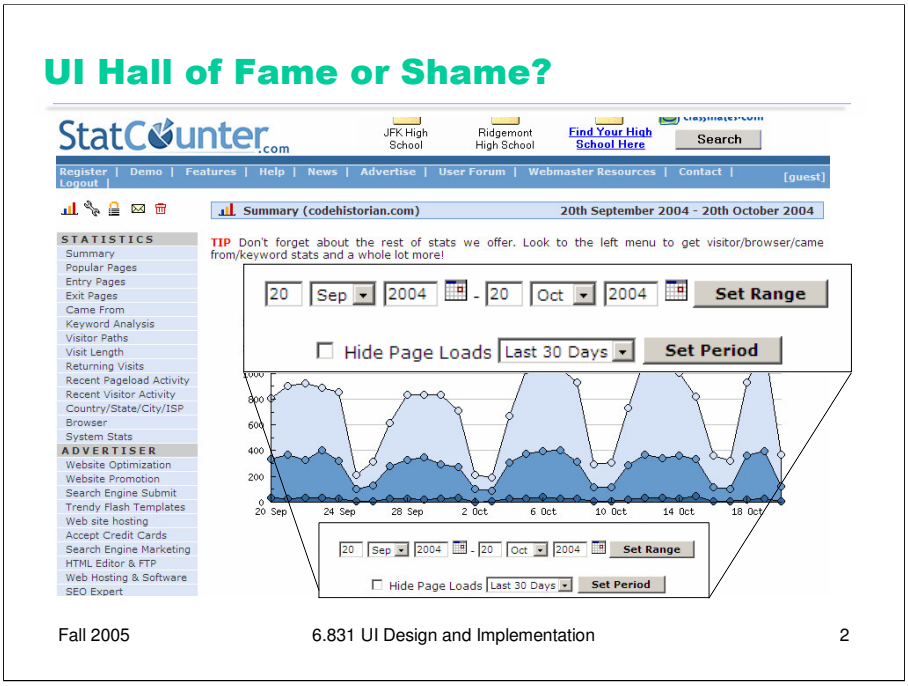
Lecture 19: Undo



Here's our hall of fame example. StatCounter is a web site that tracks usage statistics of your web site, using a hit counter. This is a sample of its statistics page, which shows you how many visitors your site had over a certain period of time.

The first thing to note is the **simplicity** of the design. Only a few colors are used: mainly a few shades of blue, plus gray and black. This simplicity allows the few unique colors – like the red TIP, and the colored tool icons – to really stand out. The design also omits unnecessary labels – for example, the date entry boxes on the bottom don't need labels like "From:" and "To:", because they're self-describing.

It's interesting to look at how **visual variables** were used to encode the information. Position is used to represent date (horizontally) and number of hits (vertically). The kind of statistic is encoded in the value of the graph line, ranging from dark blue (returning visitors) to light blue (page loads). The statistics are actually related in a subset hierarchy: since every returning visitor is a unique visitor, and every unique visitor causes at least one page load, it is always the case that page loads > unique visitors > returning visitors. This hierarchy is emphasized both by position (the curves are always in the same order vertically) and by value. Position and value were good choices for emphasizing the ordering, because both variables are ordered. An unordered visual variable, like the shape of the data point, might not have been as effective.

This page does have some problems. One is the use of two different terms, "range" and "period", which basically mean the same thing (**internal consistency**). The Set Period interface is in fact a list of common **shortcuts**, like "the last 30 days", which is a good thing; but the shortcuts should be presented more prominently. There's no reason why the year field (2004) should be a text box, rather than a drop-down with choices appropriate to the actual range of data available (**error prevention**). And the hyphen between the start date and the end date is too small to have good **contrast** with the controls around it; it disappears.

Quiz 2 next Wednesday

- Topics
  - L11: graphic design, visual variables, gestalt principles, simplicity, contrast, balance, alignment, symmetry, white space
  - L12: GUI builders
  - L13: toolkits, widgets, toolkit layering, pluggable look & feel
  - L14: heuristic evaluation, cognitive walkthrough
  - L15: animation, frame animation, palette animation, design principles, property animation, pacing & path
  - L16: user testing, formative evaluation, ethics
  - L17: alpha compositing, antialiasing, coordinate transforms, painting tricks
  - L18: predictive evaluation, keystroke level models, GOMS, CPM-GOMS
  - L19: undo design principles, selective undo, command objects
- Everything is fair game
  - Class discussion, lecture notes, readings, assignments
- Closed book exam, 80 minutes

Today's Topics

- Undo design principles
- History visualization
- Selective undo
- Command objects

Today's lecture concerns the design and implementation of undo mechanisms. Provision of undo is a crucial part of usability. Easy reversibility is a key principle of direct manipulation. It encourages exploration and experimentation, which are vital to the user's learning process. It makes inevitable errors less costly, and gives users a sense of safety and control. So undo is clearly very important to user interfaces. But undo isn't necessarily trivial to design, and it's challenging to implement as well.

Forming a Mental Model of Undo

- Undo = reverses the effect of an action
- Questions
  - What stream of actions will be undone?
  - How is the stream divided into units?
  - Which actions are undoable?
  - How much of the previous state is actually recovered?
  - How far back can you undo?

You may think it's obvious what the Undo command does: it reverses the effect of the user's last action. But it's not as simple as that. Undo's behavior can be mysterious. Undo is an example of a case where the system model is not well communicated by the user interface. The actions managed by Undo are not visible; there's no persistent, visual representation showing the next action to be undone. (Not quite true: in well-designed interfaces, the Undo menu command's label gives a hint, like "Undo Typing" or "Undo Bold". But it's not prominent, so it doesn't particularly help a user form their mental model from ordinary use.) If you ask users to predict what effect Undo will have in some particular case, they may have no idea.

Let's look at some of the questions we should ask when we're designing an undo mechanism.

What stream of actions will be undone?

- Actions in this window? (MS Office)
- Actions in this text widget? (web browser)
- Just my actions, or everybody's? (multiuser apps)
- What about actions made by the computer?
  - MS Office AutoCorrect and AutoFormat are undoable, even though user didn't do them

Undo reverses the last action made by the user, but it's not necessarily the last one in the global stream. There is no global Undo in current GUI environments. Each application, sometimes even each widget, offers its own Undo command. A particular Undo command will only affect the action stream of the application or widget that it controls – so it will undo the last action in that application or widget's stream, which isn't necessarily the last command the user issued to the system as a whole.

Some applications use a separate action stream for each window. Microsoft Office works this way, for example. If you type something into Word document A, then type something else into Word document B, then switch back to A and invoke Undo, then A's insert will be undone – even though B's insert is the last one you actually performed.

Other applications treat each *text widget* as a separate action stream. Web browsers behave this way. Try visiting a form in a web browser, and type something into two different fields. You'll find that Undo only affects the field with the current keyboard focus, ignoring actions you made on any other fields. Changes made in other kinds of form widgets – drop-down menus or listboxes, for example – aren't added to *any* action stream.

Applications with multiple simultaneous users – such as a shared network whiteboard, where anybody can scribble on it – face the question of whether Undo should affect only your own actions, or everybody's actions. Usually, the best answer to this question is only your own actions, unless you have some kind of floor control mechanism that prevents people from working simultaneously [Abowd & Dix, "Giving undo attention," *Interacting with Computers*, v4 n3, 1992].

How is the stream divided into units?

- Lexical level
  - Mouse clicks, key presses, mouse moves
  - Nobody does it at this level
- Syntactic level
  - Commands and button presses
- Semantic level
  - Changes to application data structures (e.g., the result of an entire Format dialog)
  - This is the normal level
- Text entry is aggregated into a single action
  - But other editing commands (like Backspace) and newlines interrupt the aggregation
- What about user-defined macros?
  - Undo macro actions individually, or as a unit?

Once you've decided which stream of actions to undo, the next question is, how is the stream divided into units? This is important because Undo reverses the last unit action of the stream.

Dividing at the **lexical level** means low-level input events, so Undo might reverse the very last keyboard or mouse change. For example, if you just did a drag-and-drop, invoking Undo might undo your mouse button release, putting you back into drag-and-drop mode and allowing you to drop somewhere else. No user interface (that I know of) implements lexical Undo in a systematic way; it's not clear how to get it right (since you're not holding the button down anymore!), and it's probably not what users want.

At the **syntactic level**, you would undo commands or onscreen button presses. For menu items and toolbar buttons, this is the right thing. But if you just finished a dialog – say, using the Font dialog, or selecting a Color – then this would undo the OK button press, returning you into the dialog box. Most applications don't do it at this level either.

The **semantic level** is what most designers choose, where Undo reverses the most recent change to the backend model – whether it was caused by a simple command, like Boldface, or a complicated dialog, like Page Layout. That's great for one kind of user control and freedom, since it makes complex changes just as easy to back out of as simple changes. But what if you just completed a long wizard dialog, only to discover that it didn't do what you wanted, and you have to redo it completely?

For undoing text, individual typed characters should be **aggregated** somehow – otherwise, Undo won't be any faster than pressing Backspace. One natural way to do this might be word boundaries; but most text editors use edit commands and newlines as boundaries.

In general, the action stream should be divided into **chunks** from the user's perspective. For example, a user-defined macro is a chunk, so Undo should treat the entire macro as a unit action.

Which actions are undoable?

- User's action stream may include many actions ignored by Undo
  - Selection (of text or objects)
  - Keyboard focus (to different widgets or windows)
  - Changing viewpoint (scrolling, zooming)
  - Changing layout (opening palettes or sidebars, adjusting window sizes)
  - UI customization (adding buttons to toolbars)
- So which actions does Undo actually undo?
  - Some applications (e.g. web browsers, IDEs) Undo/Redo for the editing stream, Back/Forward for the viewpoint stream

Many actions that affect visible program state may be completely ignored by Undo. Typically these actions affect the **view**, but don't actually change the backend model. Examples include selection, keyboard focus, scrolling and zooming, window management, and user interface customizations.

Since easy reversibility can be just as helpful for view changes, some applications define new commands for them, so they can reserve Undo for reversing model changes. Web browsers are a fine example: the Back button reverses a jump in view (whether caused by loading a new page or clicking on an internal hyperlink to jump to another place in the same page). Development environments like Eclipse have borrowed this idiom for navigation in code editors; you can press Back to undo window switching and scrolling.

How much state is recovered?

- Select text, delete it, and then undo
  - Text is restored
  - But is selection restored? Cursor position?

Even if the Undo stream doesn't include all the view changes you make, how much of the view state will be restored when it reverses a model change? When you undo a text edit, for example, will the selection highlight be restored as well? Will the text cursor be put back where it was before the edit? If the text scrolls, will it be scrolled back to the same place?

How far back can you undo?

- Often a limit on history size
  - Used to be one action -- now usually hundreds, or infinite
- Does action stream persist across application sessions?
  - If so, stream must be saved to file
- Does it persist across File/Save?
  - Not in MS Office

Finally, how far back will the undo history stream go? Old Macintosh applications had only single undo – i.e., you could only undo the last action, and no farther. Thankfully, cheap memory has made deep undo history feasible and commonplace.

Even though memory no longer limits undo, the conventional model of undo still does. In most applications, Undo is a transient phenomenon, limited to a single application session. If you shut down the application, and then restart it, the undo history is erased. So you can't undo past the start of the current session.

Some applications even erase the undo history as soon as the user saves a document to disk. (Microsoft Office does this.) Presumably the reason is consistency – i.e., after you save, the model should be in the same state that it would be if you closed the application and restarted it – but it poses a serious cost on users who habitually save frequently.

Design Principles for Undo

- Visibility
  - Make sure undone effects are visible
    - e.g., scrolled into view, selected, possibly animated
- Aggregation
  - Units should be "chunks" of action stream: typed strings, dialogs, macros
- Reversibility of the Undo itself
  - Support Redo as well as Undo
  - Undo to a state where user can immediately reissue the undone command, or a variant on it
    - e.g., restore selection & cursor position
- Reserve it for model changes, not view changes
  - For consistency with other applications, reserve Undo for changes to backend data
- "Undo" is not the only way to support reversibility
  - Backspace undoes typing, Back undoes browsing, Recent Files undoes file closing, scrolling back undoes scrolling
  - Forward error recovery: using new actions to fix errors

The upshot of all these questions is that it's very hard for users to predict what Undo will do. Faced with this unpredictability, a common strategy is to press Undo until you see the effect you want to reverse actually go away, or until you realize it's gone too far without solving the problem (i.e., it's reversed an older, still-desired effect). So **visibility** of Undo's effects is a critical part of making it usable. Whenever Undo undoes a command, it should make sure that the effects of that have a visible change on the screen. If the user has changed the viewpoint (e.g. scrolling) since doing the command that is now being undone, the viewpoint should be changed back, so that it's easy to see what was reversed.

The unit actions should correspond to **chunks** of the user's interaction: whole typed words (or strings), complete dialogs, user-defined macros.

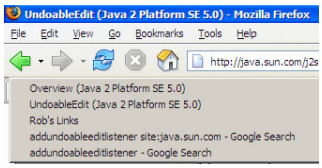
Undo itself should be reversible, so that if you overshoot, you can come back. That's what the **Redo** command is for. Another way to reverse an Undo is to manually issue the undone command again; a good undo mechanism should set up the conditions for this as well. For example, suppose you select a range of text and Delete it, and then Undo that deletion. The editor should not only restore the text, but also restore the selection highlight, so that you can immediately press Delete to delete the same text again.

For consistency, reserve the Undo command for model changes. You can use other commands for view changes. Keep in mind that you don't necessarily need a command named "Undo" to support reversibility. There are other commands that move through other action streams (Back), and physical manipulations (like scrollbar dragging) support direct reversibility.

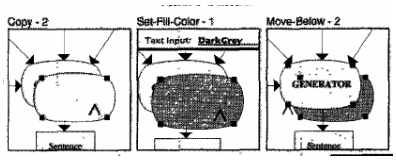
Users may not even think of reaching for Undo if the rest of your interface makes it easy to reverse undesired changes. Undo is a form of **backward error recovery**, which fixes errors by going back in time. A more natural way of thinking is **forward error recovery** – using other commands to reverse the change. For example, to undo a Bold command by forward error recovery, you select the text again and toggle Bold off. If your interface supports forward error recovery as much as possible, then warts in the Undo model won't hurt as much.

Visualizing the History

- Use Undo/Redo to browse history and view resulting application state
  - Not ideal, since user is making changes to model just to view the history
- Direct visual representation



web browser history



graphical history

The biggest usability problem with Undo is that the user can't directly see the history that they're browsing through. That's what makes it harder to learn and understand the model behind Undo.

In practice, most applications only visualize the undo history **implicitly** – i.e., the user can press Undo and Redo to browse back and forth through the history, viewing the resulting states of the entire application or document. That's hardly ideal.

Some applications use direct visualizations of history to good effect. For example, a web browser displays the history of pages visited (here, the Back button is acting as an undo command for hyperlink browsing). The browser history is concise, has user-sensible labels (page titles, not URLs), and enables direct selection (clicking on a history item to jump to it).

One research system, a drawing editor, experimented with **graphical history** – cartoon-strip visualizations of the effects of each command in the history on the actual document, zoomed in tightly to show just enough context (image from Kurlander & Feiner, "A history-based macro by example system", *UIST '92*).

Undo and Redo Lists

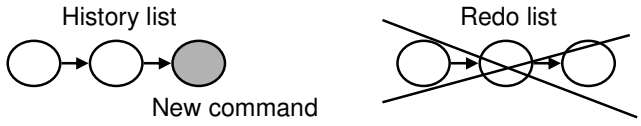
- **History list** is a **script** of commands that generates the current model state
- Undo & Redo edit the script
  - Undo removes last action from history list and puts it on **redo list**
  - Redo adds back one action from redo list
  - Undo & Redo are not put in either list

In many applications, the undo history can be formally regarded as a **script** of commands, with the invariant that the current state of the model is equivalent to the state that would be generated by running the script against the initial model state (e.g., the state of the file on disk). This model explains why applications like Microsoft Office choose to clear the undo history whenever you save the file.

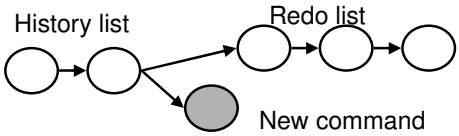
In this model, the Undo and Redo commands are not ordinary commands that are added to this script, but rather metacommands that edit the history. Undo removes the last command on the history list (and puts it at the start of a **redo list**). Redo puts the first command on the redo list back on the history list. In order to preserve the invariant, the current state of the model is changed likewise whenever the history list is changed. But the Undo and Redo commands issued by the user are *not* added to the history.

Adding New Commands to History

- New command is added to history list
  - And clears the redo list (in most apps)



- Or new command may branch history

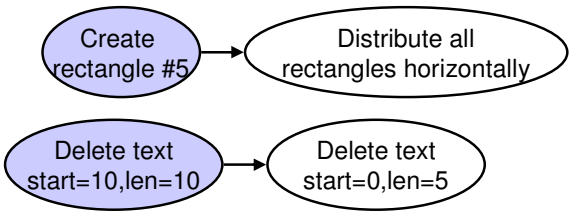


Invoking a new command usually clears the redo list. This is the safest approach, because the new command may destroy preconditions of commands sitting on the redo list. (For example, what if a command on the redo list changed the color of a certain circle, but the new command *deleted* that circle? What would redoing that command mean?)

Some research systems have experimented with a **history tree**, in which invoking a new command creates a new branch, keeping the redo list as the other branch. If the user ever backs up to that branch again in the history, the Redo command would offer a choice going forward again. Some research web browsers have adopted a similar perspective on the page history, building a tree of browsing. In practice, these models are too complicated to understand without history visualization, and even then it's not clear that they're valuable enough to be worth the complication.

Removing Commands from History (Selective Undo)

- Selective undo = deleting any action from the history list, not necessarily the last
- Selective redo = redoing any action in redo list, not necessarily the first
- Need to visualize history to choose action to undo
- Essential for multiuser applications
- Watch out for command interdependencies



Another advanced feature is **selective undo**, which allows the user to reach back and remove the effect of any action in the history, not necessarily the last action (or conversely, reach forward to any action in the redo list and apply it). Making selective undo available to the user requires some visualization of the history; otherwise users won't be able to indicate which action should be selectively undone. But multiuser applications (where users can make simultaneous changes to the same model) basically have to implement selective undo in order to support a local, per-user undo model. The application must reach back and undo this user's last action, regardless of how many changes were made by other users in the interim.

The tricky part of implementing selective undo is dependencies between commands in the history. Some examples are shown here. What happens if you selectively undo the create-rectangle action? Presumably rectangle #5 disappears -- but later in the history, the rectangle participated in an alignment operation. Do the other rectangles stay where they are, or do they behave as if the original rectangle never existed, redistributing themselves equally again? A script model of undo would dictate the latter, because of the invariant that the current state should match the result of running all the commands in the history. But a simpler model of selective undo would simply delete the rectangle (Berlage, "A Selective Undo Mechanism for Graphical User Interfaces Based on Command Objects", *TOCHI*, v1 n3, September 1994), or perhaps forbid the create-rectangle to be selectively undone.

Similarly, the representations used for commands may interfere with selective undo. Suppose the actions on a text editor's undo history are described by absolute offsets (from the start of the text). Then if you selectively undo an old action, it may corrupt the coordinates of all subsequent actions in the history. This problem can be solved by choosing a different representation (e.g., invisible markers in the text), or by implementing commutativity rules which specify how to fix up the subsequent actions in the history.

Undo Implementation

- History is a list of command objects
  - with undo() and redo() methods
- Command object must store enough data to implement undo and redo
  - Partial checkpoint of prior state
    - Move circle **from (0,0)** to (100,100)
    - Change text **from Times** to Arial
  - Object references vs. location descriptions
    - Insert "hello" at char position 33
      vs. Insert "hello" at marker #502934
  - Relative difference vs. absolute before & after values
    - Move circle from (90,120) to (100, 100)
      vs. Move circle by (10,-20)

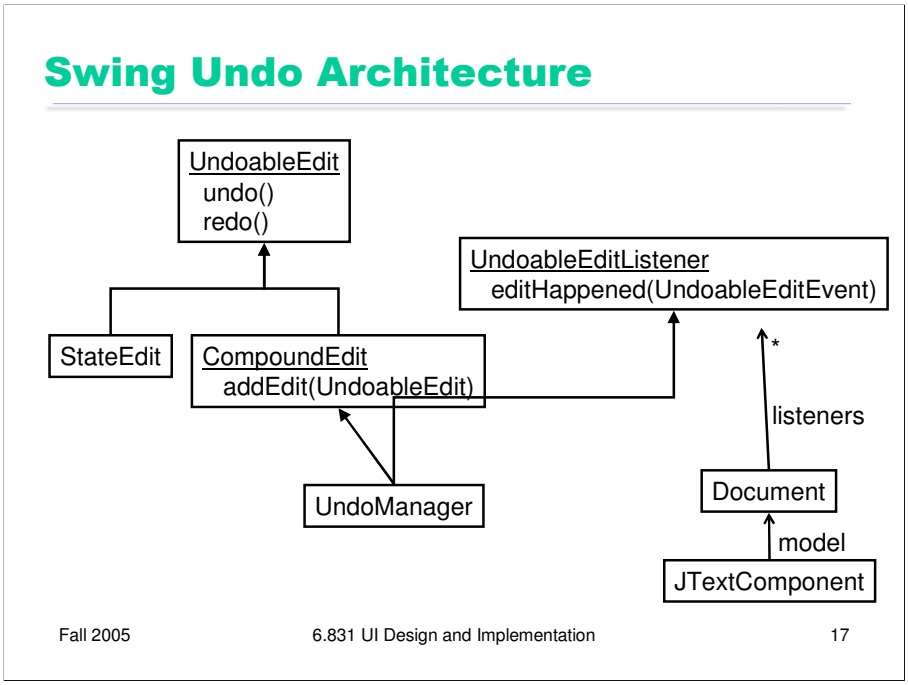
Our discussion of Undo has worked its way down from the user interface to an abstract implementation model (the script model). Now let's talk about some implementation details.

The undo history is usually represented as a list of objects, often called **command objects**, each representing a state change. These objects have undo() and redo() methods, which are called whenever the user invokes Undo or Redo to move the command objects back and forth between the history list and the redo list.

The command object must store enough information about the prior and final states of the model to implement undo and redo. In the worst case, this might have to be a **full checkpoint** of the model (e.g., in a paint program when a global filter like bluring or sharpening is applied, the command object for that filter might have to store all of the pixels in the image prior to the filter application.) More commonly, though, it's just a reference to part of the model, the prior value of that part, and the final value after the command was issued.

As we discussed on the previous slide, using of absolute location descriptions, like character positions in text, can affect the commutativity of commands, and make selective undo difficult or impossible.

An implementor might try to save space by storing only a **delta** between the before and after states. For example, a move command object might store the vector that was moved, rather than the position before and the position after. If the application supports selective undo and redo, then this decision might be noticeable to the user.



Java Swing includes a framework for managing undo in the javax.swing.undo package. In this framework, a command object implements the interface UndoableEdit, whose primary concern is providing the undo() and redo() methods. Two implementations of UndoableEdit are provided by the framework. StateEdit is a generic command object that uses a hashtable to store the prior state and final state of a model object (the model object is responsible for saving and restoring itself from this hashtable). CompoundEdit uses the composite pattern – it's a command object that contains other command objects, allowing them to be undone or redone as a unit. CompoundEdit thus supports aggregation of low-level commands into high-level chunks, like runs of typed text, dialog boxes, or user-defined macros.

The undo history for an application (or a window, or a single widget) is represented by an UndoManager, which is a CompoundEdit because it consists of a sequence of command objects. (But CompoundEdit.undo() undoes all its constituent command objects at once; UndoManager.undo() undoes them one at a time.)
The undo history implements a listener, UndoableEditListener, that receives events from a backend model. Currently, the only model in Swing that generates these events is the text document model, which is embedded inside JTextFields, JTextAreas, and JEditorPanes. But you can certainly implement your own.

Implementation Challenges

- Global changes may need to save a lot of prior state
  - e.g. whole-image operations in an image editor
- Redo of object creation must produce references usable by subsequent modification/deletion actions
  - 1: Create circle #5023 center (10,10) radius 20
  - 2: Change color circle #5023 from black to white
  - Undo 1 & 2, then redo 1
  - Redo must restore the original circle so that action 2 still refers to it
- Object references on history list prevent garbage collection of deleted objects
  - generally handled by limited history length