Undo/Redo

Principles, concepts, and Java implementation

Direct Manipulation Principles

- There is a visible and continuous representation of the domain objects and their actions. Consequently, there is little syntax to remember.
- The instruments are manipulated by physical actions, such as clicking or dragging, rather than by entering complex syntax.
- Operations are rapid and incremental
- Their effects on domain objects are immediately visible.
- Reversibility of (almost) all actions
- Users can explore without severe consequences
- Operations are self-revealing
- Syntactic correctness – every operation is legal

(from User Interface Design & Evaluation, p. 213-214)

Undo Benefits

Undo lets you recover from errors

- input errors (human) and interpretation errors (computer)
- you can work quickly (without fear)

Undo enables exploratory learning

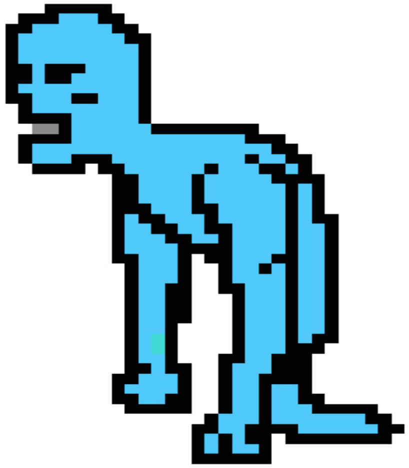
- "[In all modern user interfaces], users learn primarily by trying manipulations of visual objects rather than by reading extensive manuals." [Olsen, p. 327]
- try things you don't know the consequences of (without fear or commitment)
- try alternative solutions (without fear or commitment)

Undo lets you evaluate modifications

- fast do-undo-redo cycle to evaluate last change to document

Checkpointing

- A manual undo method
  - you save the current state so you can rollback later (if needed)
- Consider a video game …
  - You kill a monster
  - You save the game
  - You try to kill the next monster
  - You die
  - You reload the saved game
  - You try to kill the next monster
  - You kill the monster
  - You save the game
- Source code repositories are a type of check-pointing



Undo Design Choices

In any undo-redo implementation, we need to consider the following design choices.

1. Undoable actions: what can't be / isn't undone?
2. UI State restoration: what part of UI is restored after undo?
3. Granularity: how much should be undone at a time?
4. Scope: is undo/redo global in scope, local, or someplace in between?

Choice 1: Undoable Actions

Some actions may be omitted from undo:

- Change to selection? Window resizing? Scrollbar positioning?

Some actions are destructive and not easily undone:

- Quitting program with unsaved data; Emptying trash

Some actions can’t be undone:

- Printing

Undoable Actions: Suggestions

All changes to document (i.e. the model) should be undoable

Changes to the view, or the document’s interface state, should only be undoable if they are extremely tedious or require significant effort

Ask for confirmation before performing a destructive action which cannot easily be undone

- This is why you’re asked to confirm when emptying the trash!

Choice 2: UI State After Undo

What is the user interface state after an undo or redo?

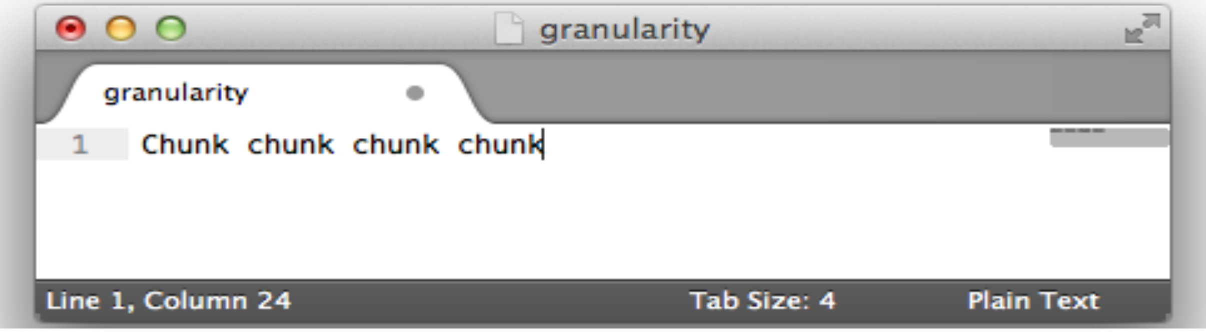
- e.g. highlight text, delete, undo … is text highlighted?
- e.g. select file icon, delete, undo … is file icon highlighted?

Suggestions:

- User interface state should be meaningful after undo/redo action is performed.
    - Change selection to object(s) changed as a result of undo/redo
    - Scroll to show selection, if necessary
    - Give focus to the control that is hosting the changed state
- Why? These provide additional undo feedback

Choice: Granularity

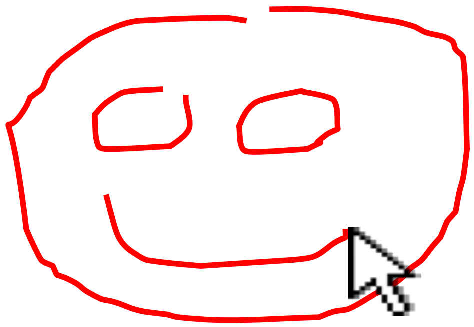
- What defines one undoable “chunk”?
  - chunk is the conceptual change from one document state to another state
- Examples
  - MS Word → string delimited by any other command (bold, mouse click, autocorrect, etc…)
  - Sublime Text Editor → token delimited by whitespace
  - Textmate Text Editor → each character
  - iOS Mail → all text since key focus



```
Chunk chunk chunk chunk
```

Example: Draw a Line

- MouseDown to start line
- MouseDrag to define line path
- MouseUp to end line
- MouseDown + MouseDrag + MouseUp = 1 conceptual chunk to “draw line”
  - “undo” should probably undo the entire line, not just a small delta in the mouse position during MouseDrags

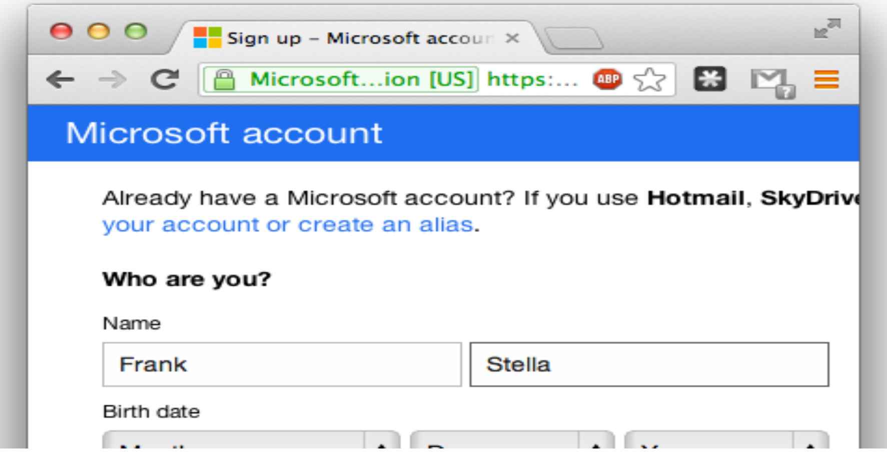


Granularity: Suggestions

- Ignore intermediate states when under continuous interactive control
  - Ex: Resizing or moving an object
  - Ex: Adjusting an image with a slider
- Chunk all changes resulting from an interface event
  - Ex: Find and replace all
  - Ex: Dialog settings
- Delimit on discrete input breaks
  - Ex: Words or sentences in text
  - Ex: Pauses in typing

Choice 3: Scope

- Is undo/redo global, local, or someplace in between?
  - – System level?
  - – Application level?
  - *– Document level?
  - – Widget level?
- Example: undo form values in Firefox vs. Chrome



Microsoft account

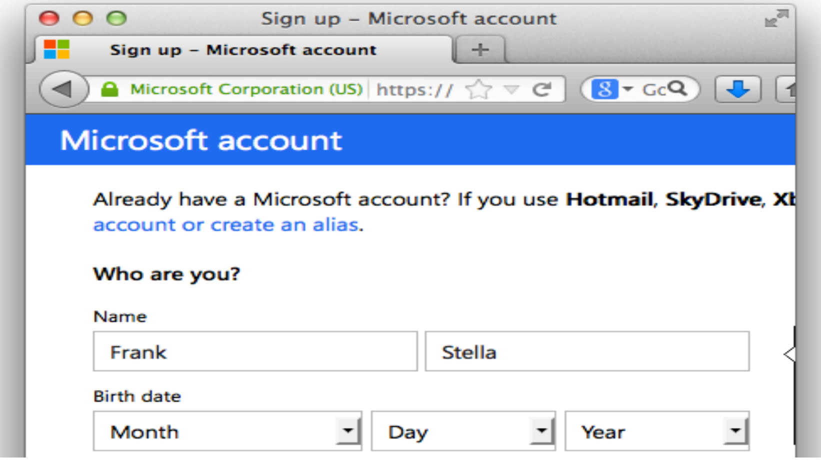
Already have a Microsoft account? If you use **Hotmail**, **SkyDrive** your account or create an alias.

Who are you?

Name

Frank | Stella

Birth date


Sign up – Microsoft account
Sign up – Microsoft account
Microsoft Corporation (US) https://

Microsoft account

Already have a Microsoft account? If you use **Hotmail**, **SkyDrive**, **X** account or create an alias.

Who are you?

Name

Frank | Stella

Birth date

Month | Day | Year

Undo Design Choices

- These are just guidelines!
  - Follow suggestions, but also **test** your undo implementation with real users.
  - You want to design behaviour that matches user’s cognitive and mental models of the system.

Implementation

Forward vs. Reverse Undo

Option 1: Forward Undo

- save complete baseline document state at some past point
- save change records to transform baseline document into current document state
- to undo last action, apply all the change records except the last one to the baseline document

Option 2: Reverse Undo

- save complete current document state
- save reverse change records to return to previous state
- to undo last action, apply last reverse change record

Change Record Implementation

Both forward and reverse undo require “change records”. We have multiple ways of implementing these as well.

CR Option 1: Memento pattern

- save snapshots of each document state
- could be complete state or difference from “last” state
- forward or reverse both just load a new document

CR Option 2: Command pattern

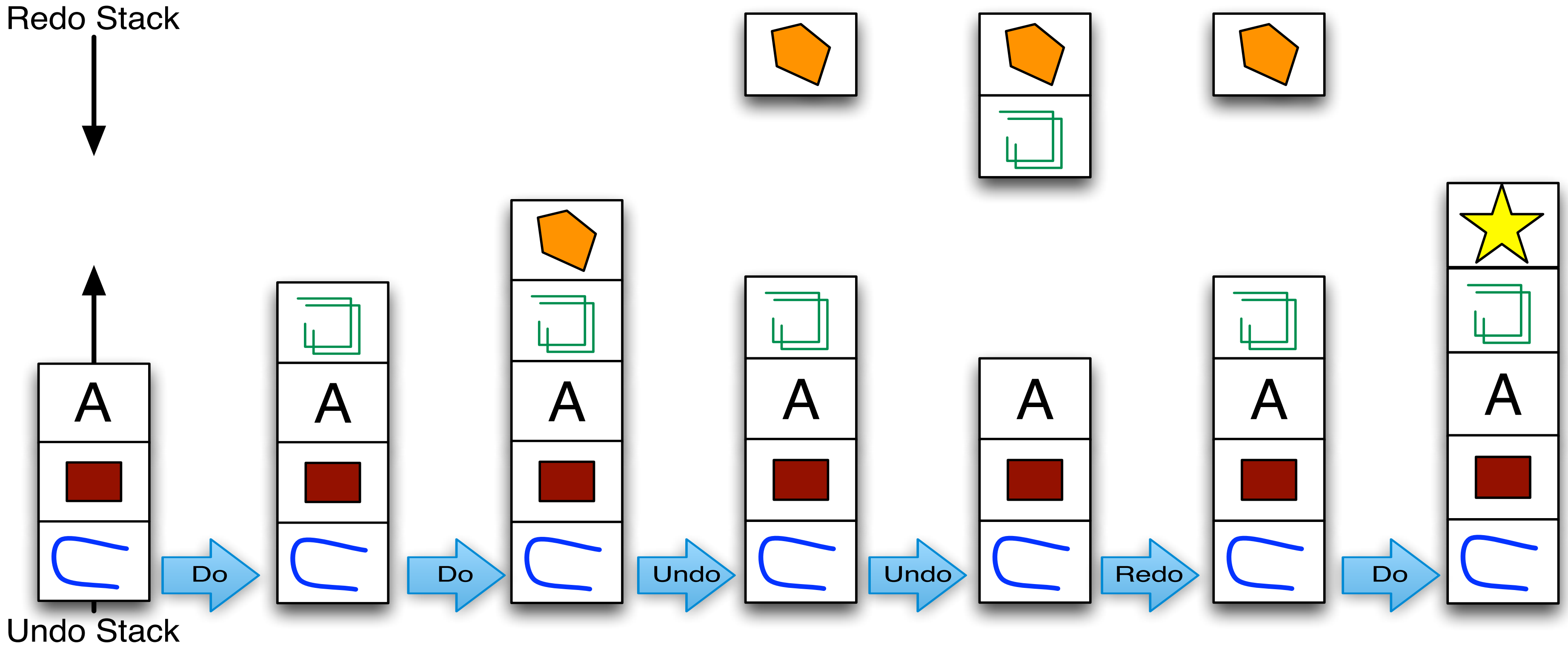
- save commands to execute (or “un-execute”) to change state

- Java uses *reverse undo with command pattern*
  - but may need Memento to save states when “information is lost”

Reverse Undo Command Pattern

- User issues command
  - – execute command to create new current document state
  - – push command onto undo stack
  - – clear redo stack
- Undo
  - – pop command from undo stack and un-execute it to create new current document state (which is the previous state)
  - – push command on redo stack
- Redo
  - – pop command off redo stack and execute it to create new current document state
  - – push on to the undo stack

Two Stacks: Undo & Redo



Ex: Text Editor Undo/Redo Commands

- insert(string, start, end)
- delete(start, end)
- bold(start, end)
- normal(start, end)

<start>	Quick brown	
<command>	Quick **brown**	`bold(6, 10)`
<command>	Quick **brown** fox	`insert(“ fox”, 11, 14)`
<undo>	Quick **brown**	`delete(11, 14)`
<undo>	Quick brown	`normal(6, 10)`
<redo>	Quick **brown**	`bold(6, 10)`
<command>	Quick **brown** dog	`insert(“ dog”, 11, 14)`

Ex: Text Editor Undo/Redo Commands

Command	Document	Undo Stack	Redo Stack
insert(“Quick brown”, 0)	Quick brown	delete(0, 10)	<empty>

bold(6, 10)	Quick **brown**	normal(6, 10)
		delete(0, 10)
			<empty>

insert(“ fox”, 11)	Quick **brown** fox	delete(11, 14)
		normal(6, 10)
		delete(0, 10)
			<empty>

undo	Quick **brown**	normal(6, 10)
		delete(0, 10)
			insert(“ fox”, 11)

undo	Quick brown	delete(0, 10)	bold(6, 10)
			insert(“ fox”, 11)

redo	Quick **brown**	normal(6, 10)
		delete(0, 10)
			insert(“ fox, 11)

insert(“ dog”, 11)	Quick **brown** dog	delete(11, 4)
		normal(6, 10)
		delete(0, 10)
			<empty>

Java Undo

Java's undo functionality in `javax.swing.undo.*`

- `UndoManager` keeps track of undo/redo command stacks
- `UndoableEdit` interface is the command to execute (redo) or un-execute (undo)

Usually put UndoManager in Model for document context

```
import javax.swing.undo.*;
// A simple model that is undoable
public class Model extends Observable {
   private int value = 0;
   // Undo manager
   private UndoManager undoManager;
   ...
}
```

Undo in Model Setters

```java
public void setValue(int v) {
    // create undoable edit
    UndoableEdit undoableEdit = new AbstractUndoableEdit() {
        final int oldValue = value;
        final int newValue = v;
        public void redo() {
            this.value = newValue; // the redo command
            notifyObservers();
        }
        public void undo() {
            this.value = oldValue; // the undo command
            notifyObservers();
        }
    };
    this.undoManager.addEdit(undoableEdit); //add edit to manager
    this.value = v; // finally, set the value
    notifyObservers();
}
```

Create an UndoableEdit and add it to the UndoManager

Triggering Undo or Redo

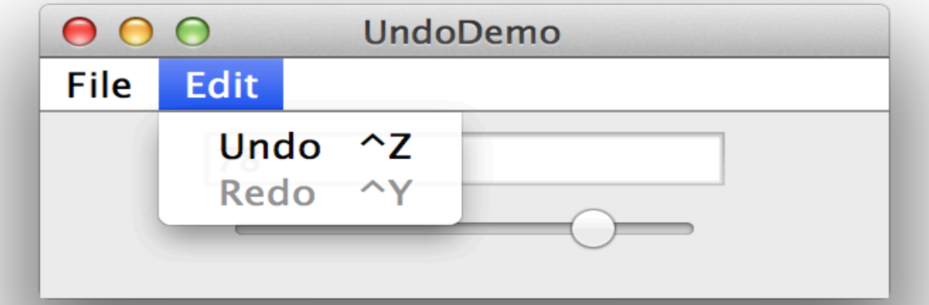
- Usually done with “undo” and “redo” menu items (with key Accelerators for CTRL-Z, CTRL-Y mapping)

```
public void undo() {
    if (undoManager.canUndo())
        undoManager.undo();
}

public void redo() {
    if (undoManager.canRedo())
        undoManager.redo();
}
```

Code Demo: UndoDemo

- Model handles all undo actions
  - UndoManager in Model
  - setters save UndoableEdits (uses closure)
  - methods added for undo state: canRedo, canUndo
- MainMenuView observes model to set enabled state for undo and redo menu items
- View doesn't know anything about undo (other than menu items); it just works
- Menu has Accelerator keys (hotkeys)

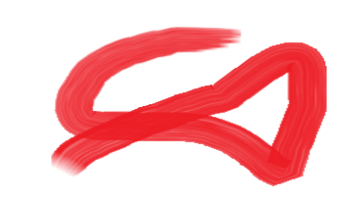
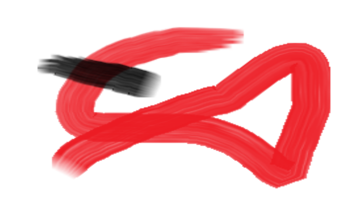
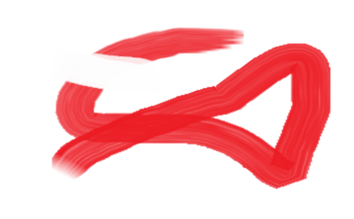


Java Undo Interfaces and Classes

- Interfaces
  - UndoableEdit: implemented by command objects. Key methods: undo, redo.
  - StateEditable: implemented by models that can save/restore their state. Key methods: storeState, restoreState
- Classes
  - AbstractUndoableEdit: convenience class for UndoableEdit
  - StateEdit: convenience class for StateEditable;
  - UndoManager: container for UndoableEdit objects (command pattern). Key methods: addEdit, canUndo, canRedo, undo, ...
  - CompoundEdit: “A concrete subclass of AbstractUndoable-Edit, used to assemble little UndoableEdits into great big ones.”

Command Undo Problems

Consider a bitmap paint application

Command	Document	Undo Stack	Redo Stack
stroke(points, 10, red)		erase(points, 10)	<empty>
stroke(points, 10, black)		erase(points, 10) erase(points, 10)	<empty>
undo		erase(points, 10)	stroke(…)

Solutions for "Destructive" Commands

Solution 1: Use forward command undo …

Solution 2: Use reverse command undo, but un-execute command stores previous state for "destructive" commands

- that's a Memento!
- might require a lot of memory
- why some applications limit the size of undo stack

- Solution 2 is commonly used in cases where it's difficult to undo a destructive action.
  - e.g. bit drawings, affine transforms.

Summary

- Benefits of undo/redo
  - enables exploratory learning
  - lets users recover from errors
  - lets users evaluate modifications (undo-redo cycle)

- Design
  - Undoable actions: what can’t be / isn’t undone?
  - UI State restoration: what part of UI is restored after undo?
  - Granularity: how much should be undone at a time?
  - Scope: is undo/redo global in scope, local?

- Implementation
  - Forward vs. Reverse undo
  - Command vs. Memento pattern