Android Tutorials

- RCH 207 @ 1:30 (120)
- MC 4060 @ 3:30 (66)
- Note: Good advice is to try the RCH session, as the MC 4060 room is small …

Undo

Most Basic Undo

- Manual undo without programmer
- 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
- Based around checkpoint/rollback
  - User manually specified a point from which to resume

Why Do We Need Anything More?

- Why offer undo?
- What does it offer us?
- How is it used by people in practice?

Use: Correcting Errors

- Fix “mistakes” in input
  - A safety net for input techniques
  - Allows faster input
  - Allows for less planning
- Two types of errors:
  - User input error (human side)
  - Interpretation error (computer side)

Use: Supporting Exploration

- "One of the key claims of direct manipulation is that users would *learn primarily by trying* manipulations of visual objects rather than by reading extensive manuals." [Olsen, p. 327]
- Exploratory learning
  - Try things you don't know the consequences of
  - Well-implemented undo can allow users to try without commitment
- Exploring alternative problem solutions
  - Again, try something without commitment
- Requirement: perceived safety

Use: Evaluation

- Fast do-undo-redo cycles
  - previous and current version are flashed in quick succession
  - provides in-place evaluation *across time*
- Examples:
  -
  -

Implementing Program Level Undo

- Choices
  - Granularity
  - Implementation
  - Context
  - Actions/Events
  - State restoration

Choices: Granularity

- What defines one undoable “operation”?
- Typing in MS Word
  - Apparently separated by a non-typing operation eg: bolding or switching to another app
- Typing in TextMate
  - A character
- Typing in TextPad
  - A line of text (always)
  - Probably because it is often used for programming where a line has a more specific meaning than in a word processor.
- Key question: What are appropriate undo “chunks”?

Choices: Granularity

- Example: drawing a free-hand line
  - User presses mouse button to begin drawing
  - User drags mouse with button pressed to define the line's path
  - User releases the mouse button at the end of the path
- Mouse down + Mouse drag + Mouse up
  - one conceptual unit
  - "undo" should probably undo the entire line, not just a small delta in the mouse position
  - mouse up defines "closure" of the conceptual unit or "operation"

Choices: Granularity

- Rules of thumb:
  - Do not record actions while actively interacting with a control.
    - Example:
  - Chunk *all* changes made in one user interface event into a single undo action.
    - Example:
  - Break input up based on discrete breaks in the input
    - Example:

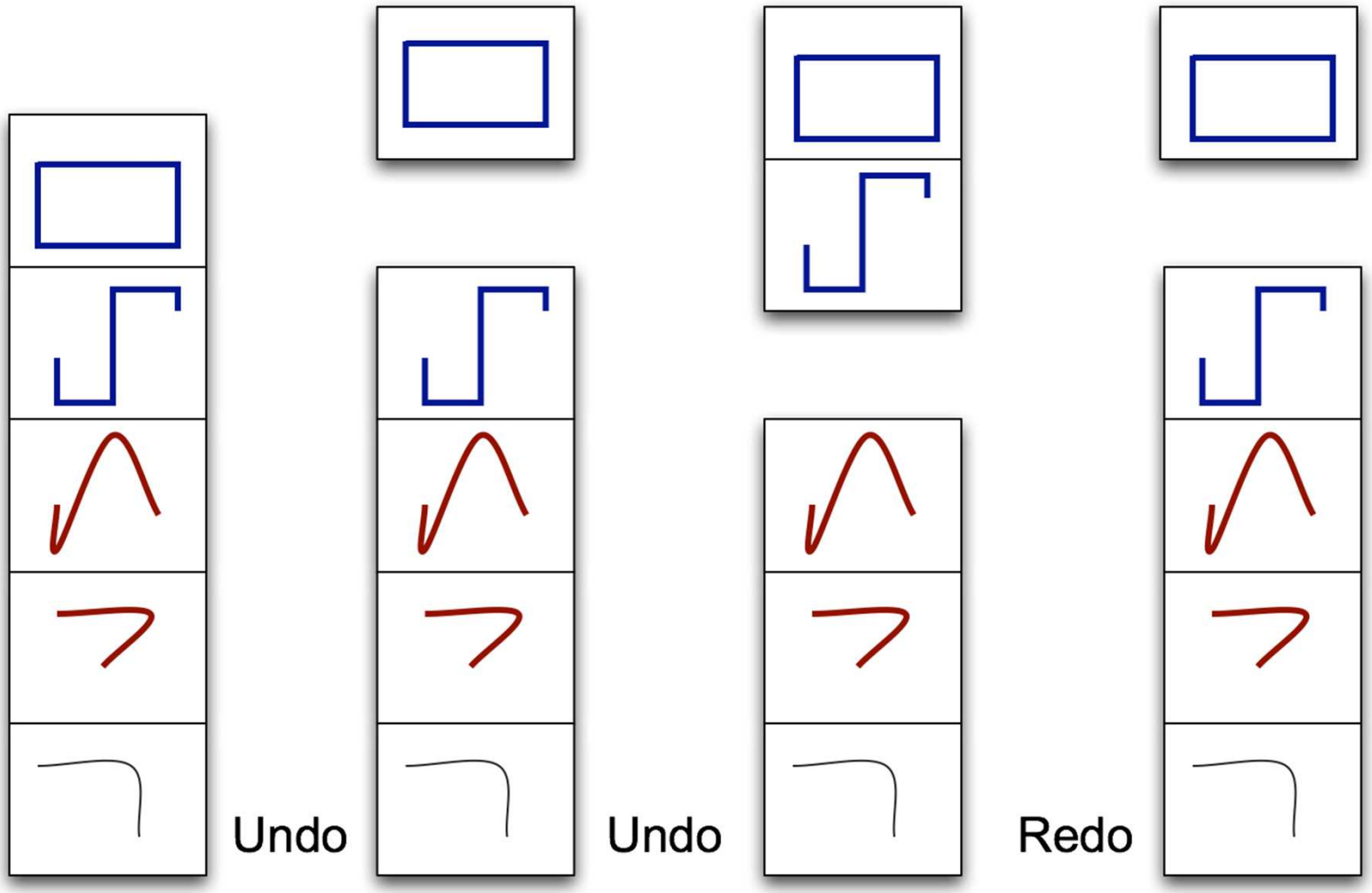
Choices: Implementation

- Need to keep a history of operations
- Undo:
  - Remove the most recent operation from the history
  - Restore the state to before the most recent operation
- Redo:
  - Reapply the most recently “undone” operation
  - Not available if there is no undone operation

Choices: Implementation

- Could imagine using either *memento* or *command* design patterns
  - Memento = save state
  - Command = analyze how to execute and un-execute commands
- Java uses command pattern, as it's slightly more memory efficient.
  - Can get debatable whether command or memento, though: think geometric transformations …
  - Assuming command pattern …

Choices: Implementation



Choices: Implementation

- Two approaches to updating the model after an undo or redo:
  - Baseline and forward undo
    - rebuild the model from a known (saved) state by reapplying each operation in a forward direction
  - Command Objects and backup undo
    - for each operation, remember how to do it and how to undo it
    - Example:

Choices: Context

- Based on the previous illustration, we need two stacks. Where should they be kept?
  - System level?
  - Application level?
  - Document level?
  - Control level?
- Example: A form in Firefox vs. a form in Safari/Chome/any WebKit-based browser
- Choices impact your underlying implementation.

Choices: Context

- Option 1: associate an undo stack with each self-contained component of the interface.
  - Example: Firefox's handling of individual text fields.
- Option 2: associate an undo stack with each document's model in the MVC architecture.
  - Implications for multi-document applications?
  - Simplified conceptual model for the user: Edits are associated with an *overall* document rather than specific controls in the user interface.

Choices: Undoable Actions

- Some things can't be undone:
  - Printing, Saving
  - Quitting program with unsaved data
  - Emptying trash
    - Ask for confirmation before doing a destructive, undoable, operation
- Some things you may choose to omit from undo, e.g.
  - Changes to selections?
  - Window resizing?
  - Scrollbar positioning?

Choices: Undoable Actions

- Rules of Thumb:
  - Any and all changes to a document's content, i.e. the *model*, should be undoable.
  - Changes to a document's *interface state* or *view* should be undoable if they are extremely tedious or require significant effort.

Choices: State Restoration

- What user interface state is restored after an undo or redo?
  - Answer: It depends on application
  - OmniGraffle versus TextPad
- Rules of Thumb:
  - 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.
- These actions help users understand the result of the undo/redo operation.

Summary: Available Choices

- Granularity: how much should be undone at a time?
- Implementation: how do you do it?
- Context: what is the scope of an undo operation?
- Undoable actions: what can't/isn't undone?
- State restoration: what UI state is restored?

- If in doubt:
  - test the implementation with real users.
  - See if they find the choices made in undo semantics intuitive in the context of their work.

Implementation in Detail

- Saving and restoring state
- Model responsibility vs. UI responsibility
- Demo Code

Impl: Saving & Restoring State

- For each operation ("chunk" of input from the user), place an object on the undo/redo stack.
- To undo the operation, pop it off the stack and execute it.
- What's the name of this Design Pattern?
- Example:
  - `someOperation.undo();`
  - `someOperation.redo();`

Impl: Saving & Restoring State

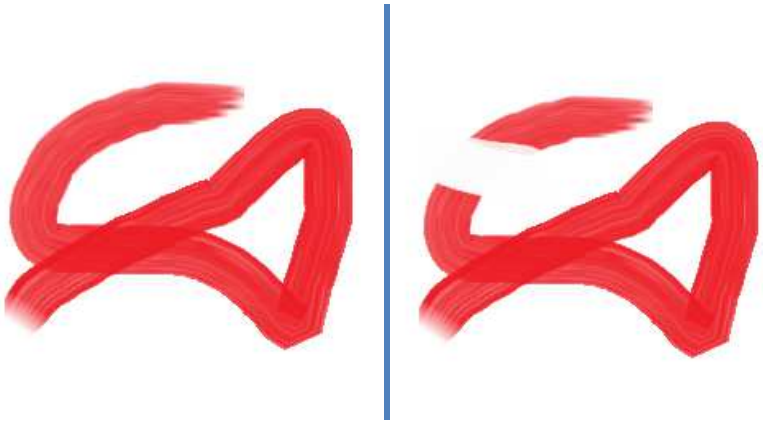
- The operation/command object restores a previous state in one of two ways:
  - Save changes to the state
  - Save the state itself

- Save changes to the state: typical in many cases
  - Word Processor
  - Vector drawing program

  - When doesn't this work?

Impl: Saving State

- Consider a bitmap painting program
  - Do red stroke
  - Do black stroke
  - Undo
- If all we do is save the command to create/remove the black stroke, what is the result?
- Need to save at least part of the image that existed before the stroke was made.
  - Might require a lot of memory!
  - MS Paint limits the number of operations you can undo => BAD



Impl: Saving & Restoring State

- If you can forward-correct an action (that is, perfectly restore from a previous state through actions alone), then just save the operations.
  - Exception: Operations that take a lot of time but don't take a lot of memory to save the change in state.
- If you cannot forward-correct an action (eg: cropping an image, paint-style drawing), you must save state so you can restore the previous state.
  - Options: store the entire state, or just the differences

Impl: Java

- Interfaces
  - StateEditable: implemented by models that can save/restore their state. Key methods: storeState, restoreState
  - UndoableEdit: implemented by command objects. Key methods: undo, redo.
- Classes
  - AbstractUndoableEdit: convenience class for UndoableEdit
  - StateEdit: convenience class for StateEditable; extends AbstractUndoableEdit. Key methods: init, end, undo, redo
  - 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."

```
public class TriangleBaseUndoableEdit extends AbstractUndoableEdit {
  private TriangleModel model;

  protected double oldBase;
  protected double newBase;

  private TriangleBaseUndoableEdit(TriangleModel model,
                double oldBase, double newBase) {
    this.model = model;
    this.oldBase = oldBase;
    this.newBase = newBase;
  }
 public void undo() {
     this.model.setBase(this.oldBase);
  }

  public void execute() {
     this.model.setBase(this.newBase);
  }

  public void redo() {
     this.execute();
  }
}
```

```
public class MenuView extends JMenuBar {
  private TriangleModel model;
  private UndoMgr undo;                  // UndoManager extended to handle observers
  private JMenu file = new JMenu("File");
  private JMenu edit = new JMenu("Edit");

  // Actions can be interpreted by menus, toolbars, etc.
  private AbstractAction newAction = new AbstractAction("New") {
    public void actionPerformed(ActionEvent e) {
      Application.getInstance().newDocument();
    }
  };

  private AbstractAction undoAction = new AbstractAction("Undo"){
    public void actionPerformed(ActionEvent e) {
      MenuView.this.undo.undo();
    }
  };
```

```
...
   // Undo manager should inform observers when a command is
   // added or it performs an undo or redo.  We then update
   // menus.
   this.undo.addObserver(new IObserve() {
     public void update(Object subject, Object detail) {
       undoAction.setEnabled(MenuView.this.undo.canUndo());
       redoAction.setEnabled(MenuView.this.undo.canRedo());
     }
   });


...
   // Set accelerator keys for the menu items.
   this.undoAction.putValue(Action.ACCELERATOR_KEY,
                KeyStroke.getKeyStroke(KeyEvent.VK_Z,
                   ActionEvent.META_MASK));
   this.redoAction.putValue(Action.ACCELERATOR_KEY,
                KeyStroke.getKeyStroke(KeyEvent.VK_Z,
                   ActionEvent.META_MASK |
                   ActionEvent.SHIFT_MASK));
 }
```

Demo 2

Undoable Widgets

Differences

- Option 1
  - Create a model observer to handle the enable/disable of Undo
  - Clean MVC architecture.
  - Works well if undo only associated with model
- Option 2
  - Integrate undo/redo into controls you activate.
  - When ActionListener is fired, state is bundled and saved in undo manager

Ideas for Improving Undo

- Branching Histories
  - Fully record every state that is visited
  - Issues
    - User may not want every state saved
    - No real elegant interfaces for browsing the histories
- Editable Histories
  - Directly edit past state; changes propagate down
  - Issue: changes made earlier in history may result in incompatible states later in the history.

Scripting

- The command objects in the undo/redo stacks can form the basis for scripting the application.
- Need methods to:
  - parse text input (eg from a file) into appropriate command objects
  - hard part is figuring out how to refer to specific parts of the model
- More about scripting using interpreters in a future lecture

Summary

- Undo-Redo is critical for:
  - Correcting errors easily
  - Exploratory learning
  - Evaluation
- There are a bunch of choices to make: granularity, implementation approach, context of each undo/redo stack, what constitutes an undoable action, and whether to undo/redo changes to interface state.
- Implementation: Command objects, where for each action we create an object that knows how to do it and to undo it, is the most common approach.