Einführung in die Programmierung
Introduction to Programming
Prof. Dr. Bertrand Meyer
October 2006 – February 2007

Lecture 21: Undo/Redo

Further reference

Chapter 21 of my Object-Oriented Software Construction, Prentice Hall, 1997
Erich Gamma et al., Design Patterns, Addison-Wesley, 1995: "Command pattern"

The problem

Enabling users of an interactive system to cancel the effect of the last command

Often implemented as "Control-Z"

Should support multi-level undo-redo ("Control-Y"), with no limitation other than a possible maximum set by the user

Why are we doing this?

Useful in every interactive application
  ➢ Don’t even think of writing an interactive system without an undo-redo mechanism
Useful design pattern ("Command")
Illustration of general algorithmic and data structure techniques
Review of O-O techniques: inheritance, deferred classes, polymorphic data structures, dynamic binding...
Beautiful and elegant

Our working example: a text editor

Notion of "current line".
Assume commands such as:
  ➢ Remove current line
  ➢ Replace current line by specified text
  ➢ Insert line before current position
  ➢ Swap current line with next if any
  ➢ "Global search and replace" (hereafter GSR): replace every occurrence of a specified string by another
  ➢ ...

This is a line-oriented view for simplicity, but the discussion applies to more sophisticated views
Underlying class (from application*)

```java
class EDIT_CONTROLLER

  feature

  text: LINKED_LIST[STRING]

  remove is
  -- Delete line at current position.
  require
  do not off
  end
  text.remove

  put_right (line: STRING) is
  -- Insert line after current position.
  require not after
  do text.put_right(line)
  end

end
```

A straightforward solution

- Before performing any operation, save entire state
  - In the example: text being edited, current position in text
- If user issues "Undo" request, restore entire state as last saved

But: huge waste of resources, space in particular

Intuition: only save the "diff" between states.

Key step in devising a software architecture

**Finding the right abstractions**

(the interesting object types)

Here:

The notion of "command"

Keeping the history of the session

The history list:

```
<table>
<thead>
<tr>
<th>Insertion</th>
<th>Insertion</th>
<th>Removal</th>
<th>Insertion</th>
<th>Swap</th>
</tr>
</thead>
<tbody>
<tr>
<td>Oldest</td>
<td>Most recent</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

history: LINKED_LIST [COMMAND]

What's a “command” object?

A command object includes information about one execution of a command by the user, sufficient to:

- Execute the command
- Cancel the command if requested later

For example, in a Removal command object, we need:

- The position of the line being removed
- The content of that line!

General notion of command

```java
defered class COMMAND

  feature

  done: BOOLEAN

  deferred
  -- Has this command been executed?

  execute is
  -- Carry out one execution of this command.
  deferred
  ensure already: done

  undo is
  -- Cancel an earlier execution of this command.
  require already: done
  deferred
  end

end
```
Command class hierarchy

```
COMMAND
  execute
  undo

REMOVAL
  execute
  undo
  line: STRING
  index: INTEGER

INSERTION
  execute
  undo

... deferred
... effective
```

A command class (sketch, no contracts)

```
class REMOVAL inherit COMMAND feature
  controller: EDIT_CONTROLLER -- Access to business model
  line: STRING -- Line being removed
  index: INTEGER -- Position of line being removed
  execute is -- Remove current line and remember it.
    do
      line := controller.item ; index := controller.index
      controller.remove ; done := True
    end
  undo is -- Re-insert previously removed line.
    do
      controller.go_ith (index)
      controller.put_left (line)
    end
end
```

Underlying class (from business model)

```
class EDIT_CONTROLLER feature
  text: LINKED_LIST [STRING]
  remove is -- Remove line at current position.
    require not off
    do
      text.remove
    end
  put_right (line: STRING) is -- Insert line after current position.
    require not after
    do
      text.put_right (line)
    end
end
```

The history list

A polymorphic data structure:

```
Insertion Insertion Removal Insertion Swap

Oldest Most recent
```

Reminder: the list of figures

```
class LIST[G]
  feature
    last: G is ...
    extend (x: G) is ...
    ...
fl: LIST [FIGURE]
  r: RECTANGLE
  s: SQUARE
  t: TRIANGLE
  p: POLYGON
  ...
fl.extend (p); fl.extend (t); fl.extend (s); fl.extend (r)
fl.last.display
```

The history list

A polymorphic data structure:

```
Insertion Insertion Removal Insertion Swap

Oldest Most recent
```

history: LINKED_LIST [COMMAND]
Executing a user command

decode_user_request
if "Request is normal command" then
    "Create command object c corresponding to user request"
    history.extend(c)
    c.execute
else if "Request is UNDO" then
    if not history.before then -- Ignore excessive requests
        history.item.undo
    end
else if "Request is REDO" then
    if not history.is_last then -- Ignore excessive requests
        history.forth
        history.item.execute
    end
end

Conditional creation (1)

A
B
C
D

a1 : A
if condition_1 then
    -- "Create a1 as an instance of B"
    create b1.make (...) a1 := b1
else if condition_2 then
    -- "Create a1 as an instance of C"
    create c1.make (...) a1 := c1
... etc.

Conditional creation (2)

a1 : A
if condition_1 then
    -- "Create a1 as an instance of B"
    create b1.make (...) a1 := b1
else if condition_2 then
    -- "Create a1 as an instance of C"
    create c1.make (...) a1 := c1
... etc.

Creating command objects: first approach

c : COMMAND
...
decode_user_request
if "Request is remove" then
    create (REMOVAL) c
else if "Request is insert" then
    create (INSERTION) c
... etc.

Command class hierarchy

COMMAND
  +-----+-----+
  |     |     |
  |   *  |   +  |
  | deferred | effective |
  +-----+-----+
  | REMOVAL | INSERTION |
  +-----+-----+
  | execute* | execute*
  | undo* | undo*
  | line | index
  | ... | ...
Creating command objects: better approach

Give each command type a number

Initialy, fill an array commands with one instance of every command type.

To get a new command object:

\[ c := \text{commands[command_type].twin} \]

Determine command_type

Removal

Insertion

Swap

\[ 1 \quad n \]

Duplicate a "prototype"

The undo-redo (or "command") pattern

Has been extensively used (e.g. in EiffelStudio and other Eiffel tools)

Fairly easy to implement

Details must be handled carefully (e.g. some commands may not be undoable)

Elegant use of O-O techniques

Disadvantage: explosion of small classes

Preview: using agents

For each user command, have two routines:

- The routine to do it
- The routine to undo it!

The history list in the undo-redo pattern

\[ \text{history: LINKED\_LIST [COMMAND]} \]

Insertion

Removal

Insertion

Swapping

Oldest

Most recent

The history list using agents

The history list simply becomes a list of agent pairs:

\[ \text{history: LINKED\_LIST [TUPLE] [PROCEDURE \{ANY, TUPLE\}, PROCEDURE \{ANY, TUPLE\}]} \]

Insertion

Removal

Insertion

Swap

Basic scheme remains the same, but no need for command objects any more; the history list simply contains agents.

Executing a user command (before)

\[
\text{decode_user_request}
\]

\[
\text{if "Request is normal command" then}
\]

\[
\quad \text{create command object } c \text{ corresponding to user request}
\]

\[
\quad \text{history.extend}(c)
\]

\[
\quad c.\text{execute}
\]

\[
\text{elseif "Request is UNDO" then}
\]

\[
\quad \text{if not history.before then -- Ignore excessive requests}
\]

\[
\quad \quad \text{history.item.undo}
\]

\[
\quad \text{history.back}
\]

\[
\text{end}
\]

\[
\text{elseif "Request is REDO" then}
\]

\[
\quad \text{if not history.is_last then -- Ignore excessive requests}
\]

\[
\quad \text{history.forth}
\]

\[
\quad \text{history.item.execute}
\]

\[
\text{end}
\]
Executing a user command (now)

"Decode user_request giving two agents do_it and undo_it"
if "Request is normal command" then
    history.extend([do_it, undo_it])
do_it.call() 
elseif "Request is UNDO" then
    if not history.before then
        history.item.item(2).call()
        history.back
    end
elseif "Request is REDO" then
    if not history.is_last then
        history.forth
        history.item.item(1).call()
    end
end

What we have seen

People make mistakes!
Even when they don't mess up, they want to experiment:
undo-redo supports "trial and error" experimental style

Undo-redo pattern:
> Very useful in practice
> Widely used
> Fairly easy to implement
> Excellent illustration of elegant O-O techniques
> Even better with agents!
(see next lectures...)

Further reference

Chapter 21 of "Object-Oriented Software Construction",
Prentice Hall, 1997

Reading assignment for next Monday

Touch of Class, chapters on agents and event-driven
design

End of lecture 21