Introduction to Programming

Bertrand Meyer

Last revised 20 January 2004
Lecture 23: An example: undo-redo
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, with no limitation other than a possible maximum set by the user.
A text editor

Notion of “current line”. Assume commands such as:

- Insert line after current position
- Insert line before current position
- Delete current line
- Replace current line
- Swap current line with next if any
- ...

This is a line-oriented view for simplicity, but the discussion applies to more sophisticated views.
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
Key step in devising a software architecture

Finding the right abstractions

(The types of objects involved.)

Here:

- The notion of “command”
Keeping the history of the session

The history list:

```
history: LINKED_LIST [COMMAND]
```

Oldest Most recent
What’s a “command” object?

A command object (instance of the class `COMMAND`) 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 delete command (as implemented by `remove`), we need:

- The position of the line being deleted
- The content of that line!
deferred class COMMAND feature

execute is
  -- Carry out one execution of this command.
  deferred
end
undo is
  -- Cancel an earlier execution of this command.
  deferred
end
A general notion of command

defered class COMMAND feature

done: BOOLEAN is
    -- Has this command been executed?

execute is
    -- Carry out one execution of this command.

defered ensure
done: done
end

undo is
    -- Cancel an earlier execution of this command.

require
    already: done
defered done
end
Command class hierarchy

- COMMAND
  - execute*
  - undo*

- DELETION
  - execute^+
  - undo^+
  - line
  - index
  - ...

- INSERTION
  - execute^+
  - undo^+
  - index
  - ...

- deferred
+ effective

...
A command class (sketch, no contracts)

class DELETION inherit COMMAND feature

    controller: EDIT_CONTROLLER
        -- Access to business model

    line: STRING
        -- The line being deleted

    index: INTEGER
        -- Position of line being deleted

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
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

  ... Also item, index, go_ith, put_left ...

end
Executing a user command

The history list:

```
history: LINKED_LIST [COMMAND]
```
decode_user_request

if "Request is normal command" then
    -- "Create command object c corresponding to user request"
    history.extend (c)
    c.execute
elseif "Request is UNDO" then
    if not history.before then
        history.item.undo
        history.back
    end
elseif "Request is REDO" then
    if not history.is_last then
        history.forth
        history.item.undo
    end
end
end
Conditional creation (1)

\[ a1: A \]

\textbf{if condition\_1 then}
\begin{itemize}
  \item \textquote{Create \(a1\) as an instance of \(B\)}
\end{itemize}
\textbf{elseif condition\_2 then}
\begin{itemize}
  \item \textquote{Create \(a1\) as an instance of \(C\)}
\end{itemize}
\textbf{... etc...}

\[ a1: A; b1: B; c1: C; d1: D; ... \]

\textbf{if condition\_1 then}
\begin{itemize}
  \item \texttt{create b1.make (...) \(a1 := b1\)}
\end{itemize}
\textbf{elseif condition\_2 then}
\begin{itemize}
  \item \texttt{create c1.make (...) \(a1 := c1\)}
\end{itemize}
\textbf{... etc...}
A

B

C

D

\[ a1: A \]

\[ \text{if } \text{condition}_1 \text{ then} \]
\[ \quad \text{-- “Create } a1 \text{ as an instance of } B \” \]
\[ \text{elseif } \text{condition}_2 \text{ then} \]
\[ \quad \text{-- “Create } a1 \text{ as an instance of } C \” \]
\[ \quad \text{... etc...} \]

\[ a1: A \]

\[ \text{if } \text{condition}_1 \text{ then} \]
\[ \quad \text{create } \{B\} a1.\text{make}(\ldots) \]
\[ \text{elseif } \text{condition}_2 \text{ then} \]
\[ \quad \text{create } \{C\} a1.\text{make}(\ldots) \]
\[ \quad \text{... etc...} \]
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

elseif “Request is UNDO” then
    if not history.before then
        history.item.undo
        history.back
        -- Ignore excessive requests
    end

elseif “Request is REDO” then
    if not history.is_last then
        history.forth
        history.item.undo
        -- Ignore excessive requests
    end
end
Creating command objects (1)

c: COMMAND

... etc...

decode_user_request

if “Request is delete” then
    create {DELETION} c

elseif “Request is insert” then
    create {INSERTION} c

... etc...
Command class hierarchy

- **COMMAND**
  - * deferred
  - + effective

- **DELETION**
  - execute*
  - undo*
  - + deferred
  - + effective
  - line
  - index

- **INSERTION**
  - execute*
  - undo*
  - + deferred
  - + effective
  - index
  - ...

- ...
Creating command objects (2)

- Give each command type a number (or other key)
- Initially, fill in a table (e.g. an array), with one instance of each command type.

- To get a new command object:

  “Determine command_type”
  \( c := \text{clone (COMMAND\_TABLE.item (command\_type))} \)
The undo-redo pattern

- Has been extensively used (e.g. in Eiffel tools)
- Fairly easy to implement
- Elegant use of O-O techniques
- Disadvantage: explosion of small classes

- In Java, can use “inner” classes.
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

**history: LINKED_LIST [COMMAND]**

- Insert
- Insert
- Remove
- Insert
- Swap

**Oldest** → **Most recent**
The history list simply becomes a list of agents pairs:

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

Basic scheme remains the same, but no need for command objects any more; we simply store agents.

This is the technique used in all current Eiffel tools
decode_user_request
if "Request is normal command" then
-- "Create command object c corresponding to user request"
history.extend (c)
c.execute
elseif "Request is UNDO" then
if not history.before then
  history.item.undo
  history.back
-- Ignore excessive requests
end
elseif "Request is REDO" then
if not history.is_last then
  history.forth
  history.item.undo
-- Ignore excessive requests
end
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
    if not history.item.item (2).call ([]) 
    history.back 
  end 

elseif “Request is REDO” then
  if not history.is_last then
    if not history.forth then
      history.item.item (1).call ([]) 
  end 
end 
end
End of lecture 23