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*)

```plaintext
class EDIT_CONTROLLER feature
  text: LINKED_LIST(STRING)
  remove is
    require not off
do
  end
  text.remove
  put_right(line: STRING) is
    require not after
do
  end
  text.put_right(line)
end
```

A straightforward solution

1. Before performing any operation, save entire state
   - In the example: text being edited,
     current position in text
2. 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:

- Insertion
- Insertion
- Removal
- Insertion
- Swap

Oldest Most recent

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

def default class COMMAND feature


done: BOOLEAN
  -- Has this command been executed?

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

deferred ensure
  already: done
end

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

**Underlying class (from business model)**

```plaintext
class EDIT_CONTROLLER feature
  text: LINKED_LIST [STRING]
  remove
    require -- Remove line at current position.
    do
      text.remove
    end
  put_right (line: STRING) is
    require -- Insert line after current position.
    do
      text.put_right (line)
    end
end
```

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

```plaintext
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
    do
      line := controller.item; index := controller.index
      controller.remove; done := True
    end
  undo is
    do
      controller.go_ith (index)
      controller.put_left (line)
    end
end
```
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 "Request is UNDO" then
  if not history.before then -- Ignore excessive requests
    history.item.undo
    history.back
  end
else "Request is REDO" then
  if not history.is_last then -- Ignore excessive requests
    history.forth
    history.item.execute
  end
end
```

Pseudocode, see implementation next

Conditional creation (1)

```
a1 : A
if condition_1 then
  -- "Create a1 as an instance of B"
  elseif condition_2 then
  -- "Create a1 as an instance of C"
  ... etc.
```

```
a1 : A
if condition_1 then
  create b1.make (...) a1 := b1
elseif condition_2 then
  create c1.make (...) a1 := c1
  ... etc.
```

Conditional creation (2)

```
a1 : A
if condition_1 then
  -- "Create a1 as an instance of B"
  elseif condition_2 then
  -- "Create a1 as an instance of C"
  ... etc.
```

```
a1 : A
if condition_1 then
  create {B} a1.make (...) 
elseif condition_2 then
  create {C} a1.make (...) 
  ... etc.
```
Executing a user command

```plaintext
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
        history.back
    end
else if "Request is REDO" then
    if not history.is_last then
        -- Ignore excessive requests
        history.forth
        history.item.execute
    end
end
```

Creating command objects: first approach

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

Command class hierarchy

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

Give each command type a number
Initially, fill an array `commands` with one instance of every command type.

To get a new command object:

```
"Determine command_type"
```

```
c := (commands [command_type]).twin
```

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

`history : LINKED_LIST [COMMAND]

<table>
<thead>
<tr>
<th>Insertion</th>
<th>Insertion</th>
<th>Removal</th>
<th>Insertion</th>
<th>Swap</th>
</tr>
</thead>
</table>

Oldest | Most recent

The history list using agents

The history list simply becomes a list of agents pairs:

`history : LINKED_LIST [TUPLE

[PROCEDURE [ANY, TUPLE],
   PROCEDURE [ANY, TUPLE]]`

<table>
<thead>
<tr>
<th>Insertion</th>
<th>Insertion</th>
<th>Removal</th>
<th>Insertion</th>
<th>Swap</th>
</tr>
</thead>
</table>

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

Executing a user command (before)

```
declare_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 -- Ignore excessive requests
        history.before
    history.item.undo
    history.back
else
    if history.is_last then -- Ignore excessive requests
        history.item.execute
```
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 ()
else if "Request is UNDO" then
    if not history.before then
        history.item.item (2).call ()
        history.back
    end
else if "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" experimenatal 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