21

Inheritance case study: "undo" in an interactive system

For our second design example we examine a need that confronts the designers of almost any interactive system: how to provide a way to undo commands.

The discussion will show how inheritance and dynamic binding yield a simple, regular and general solution to an apparently intricate and many-faceted problem; and it will teach us a few general lessons about the issues and principles of object-oriented design.

21.1 PERSEVERARE DIABOLICUM

To err is human, it is said, but to foul things up for good takes a computer (aided, one should add, by humans). The faster and more powerful our interactive systems become, the easier it becomes to make them perform actions that we do not really want. This is why we all wish for a way to erase the recent past; not the "big red button" of computer jokes, but a Big Green Button that we can push to pretend that we did not do something that we did but wish we did not.

Undoing for fun and profit

In an interactive system, the equivalent of the Big Green Button is an Undo operation, which the system's designer has provided for the benefit of any user who, at some stage in a session, wants to cancel the effect of the last executed command.

The primary aim of an undo mechanism is to allow users to recover from potentially damaging input mistakes. It is all too easy to type the wrong character or click on "OK" instead of "Cancel". But a good undo facility goes further. It frees users from having to concentrate nervously on every key they type and button they click. Beyond this, it encourages a "*What if*… ?" style of interaction in which users try out various sorts of input, knowing that they can back up easily if the result is not what they expect.

Every good interactive system should provide such a mechanism. When present, it tends to be one of the most frequently used operations. (For that reason, the makers of the computer on my desk have wisely provided an Undo key on the keyboard, although it is neither green nor particularly big. It is only effective, of course, for those regrettably few software applications whose authors took notice of it.)

Multi-level undo and redo

Offering an undo mechanism is better than not offering one, but it is not enough. Most systems that provide Undo limit themselves to one level: you can only cancel the effect of the last command. If you never make two mistakes in a row, this is enough. But if you ever go off in the wrong direction, and wish you could go back several steps, you are in trouble. (Anyone having used Microsoft Word, the Unix Vi editor or FrameMaker, in the releases available at the time this book was published, will know exactly what I mean.)

There is really no excuse for the restriction to one level of undoing. Once you have set up the undoing machinery, going from one-level to multi-level undo is a simple matter, as we will see in this chapter. And, please (this is a potential customer speaking) do not, like so many application authors, limit the number of commands that can be undone to a ridiculously small value; if you must limit it at all, let the user choose his own limit (through a “preferences” setting that will apply to all future sessions) and set default to at least 20. The overhead is small if you apply the techniques below, and is well justified.

With multi-level undo, you will also need a Redo operation for users who get carried away and undo too much. With one-level undo no special Redo is required; the universally applied convention is that an Undo immediately following an Undo cancels it, so that Redo and Undo are the same operation. But this cannot work if you can go back more than one step. So we will have to treat Redo as a separate operation.

Practical issues

Although undo-redo can be retrofitted with reasonable effort into a well-written O-O system, it is best, if you plan to support this facility, to make it part of the design from the start — if only because the solution encourages a certain form of software architecture (the use of *command classes*) which, although beneficial in other respects, does not necessarily come to mind if you do not need undoing.

To make the undo-redo mechanism practical you will have to deal with a few practical concerns.

First you must include the facility in the user interface. For a start, we may just assume that the set of operations available to users is enriched with two new requests: Undo (obtained for example by typing control-U, although following the Macintosh convention control-Z seems to have become the standard on PC tools) and Redo (for example control-R). Undo cancels the effect of the last command not yet undone; Redo re-executes the last undone command not yet redone. You will have to define some convention for dealing with attempts to undo more than what has been done (or more than what is remembered), or to redo more than what has been undone: ignore the request, or bring up a warning message.

This is only a first shot at user interface support for undo-redo. At the end of this chapter we will see that a nicer, more visual interface is possible.

Second, not all commands are undoable. In some cases this is an impossibility of fact, as in the command "fire the missiles" (notwithstanding the televised comment of a then-in-office US president, who thought one could command a U-turn) or, less ominously, "print the page". In other cases, a command is theoretically undoable but the overhead is not worth the trouble; text editors typically do not let you undo the effect of a Save command, which writes the current document state into a file. The implementation of undoing will need to take into account such non-undoable commands, making this status clear in the user interface. Be sure to restrict non-undoable commands to cases for which this property is easily justifiable in user terms.

As a counter-example, a document processing tool which I frequently use tells its user, once in a while, that in the current state of the document the command just requested is not undoable, with no other visible justification than the whim of the program. At least it says so in advance — in most cases.

Interestingly, this warning is in a sense a lie: you *can* undo the effect if you want, although not through Undo but through "Revert to last saved version of the document". This observation yields a user interface rule: if there remains any case for which you feel justified to make a command non-undoable, do not follow the document processing system's example by just displaying a warning of the form "This command will not be undoable" and giving the choice between **Continue anyway** and **Cancel**. Give users *three* possibilities: save document, then execute command; execute without saving; cancel.

Finally, it may be tempting to offer, besides Undo and Redo, the more general "Undo, Skip and Redo" scheme, allowing users, after one or more Undo operations, to skip some of the commands before triggering Redo. The user interface shown at the end of this chapter could support this extension, but it raises a conceptual problem: after you skip some commands, the next one may not make sense any more. As a trivial example assume a text editor session, with a text containing just one line, and a user who executes the two commands

(1) Add a line at the end.
(2) Remove the second line.

Exercise E21.4, page 716.

Our user undoes both, then wants to skip (1) and redo (2). Unfortunately at this stage (2) is meaningless: there is no second line. This is less a problem in the user interface (you could somehow indicate to the user that the command is impossible) than in the implementation: the command Remove the second line was applicable to the object structure obtained as a result of (1), but applying it to the object structure that exists prior to (1) may be impossible (that is to say, cause a crash or other unpleasant results). Solutions are certainly possible, but they may not be worth the trouble.

Requirements on the solution

The undo-redo mechanism that we set out to provide should satisfy the following properties.

U1 • The mechanism should be applicable to a wide class of interactive applications, regardless of the application domain.

U2 • The mechanism should not require redesign for each new command.

U3 • It should make reasonable use of storage.

U4 • It should be applicable to both one-level and arbitrary-level Undo.

The first requirement follows from the observation that there is nothing application-specific about undoing and redoing. To facilitate the discussion, we will use as example a kind of tool familiar to everyone: a text editor (such as Notepad or Vi), which enables its users to enter texts and to perform such commands as INSERT_LINE, DELETE_LINE, GLOBAL_REPLACEMENT (of a word by another) and so on. But this is only an example and none of the concepts discussed below is specific to text editors.

The second requirement excludes treating Undo and Redo as just any other command in the interactive system. Were Undo a command, it would need a structure of the form

```
if “Last command was INSERT_LINE” then
    “Undo the effect of INSERT_LINE”
elseif “Last command was DELETE_LINE” then
    “Undo the effect of DELETE_LINE”
etc.
```

We know how bad such structures, the opposite of what the Single Choice principle directs us to use, are for extendibility. They have to be changed every time you add a command; furthermore, the code in each branch will mirror the code for the corresponding command (the first branch, for example, has to know a lot about what INSERT_LINE does), pointing to a flawed design.

See “Single Choice”, page 61.

The third requirement directs us to be sparing in our use of storage. Supporting undo-redo will clearly force us to store *some* information for every Undo; for example when we execute a DELETE_LINE, we will not be able to undo it later unless we put aside somewhere, before executing the command, a copy of the line being deleted and a record of its position in the text. But we should store only what is logically necessary.

The immediate effect of this third requirement is to exclude an obvious solution: saving the whole system state — the entire object structure — before every command execution; then Undo would just restore the saved image. This would work but is terribly wasteful of space. Too bad, since the solution would be trivial to write: just use the *STORABLE* facilities for storing and retrieving an entire object structure in a single blow. But we must look for something a little more sophisticated.

On STORABLE see “Deep storage: a first view of persistence”, page 250.

The final requirement, supporting an arbitrary depth of undoing, has already been discussed. It will turn out to be easier to consider a one-level mechanism first, and then to generalize it to multi-level.

These requirements complete the presentation of the problem. It may be a good idea, as usual, to spend a little time looking for a solution on your own before proceeding with the rest of this chapter.

21.2 FINDING THE ABSTRACTIONS

The key step in an object-oriented solution is the search for the right abstraction. Here the fundamental notion is staring us in the eyes.

Command as a class

The problem is characterized by a fundamental data abstraction: *COMMAND*, representing any editor operation other than Undo and Redo. Execution is only one of the features that may be applied to a command: the command might be stored, tested — or undone. So we need a class of the provisional form

```
deferred class COMMAND feature
    execute is deferred end
    undo is deferred end
end
```

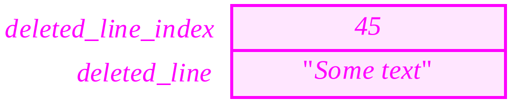
COMMAND describes the abstract notion of command and so must remain deferred. Actual command types are represented by effective descendants of this class, such as

```
class LINE_DELETION inherit
    COMMAND
feature
    deleted_line_index: INTEGER
    deleted_line: STRING
    set_deleted_line_index (n: INTEGER) is
            -- Set to n the number of next line to be deleted.
        do
            deleted_line_index := n
        end
    execute is
            -- Delete line.
        do
            "Delete line number deleted_line_index"
            "Record text of deleted line in deleted_line"
        end
    undo is
            -- Restore last deleted line.
        do
            "Put back deleted_line at position deleted_line_index"
        end
end
```

And similarly for each command class.

What do such classes represent? An instance of *LINE_DELETION*, as illustrated below, is a little object that carries with it all the information associated with an execution of the command: the line being deleted (*deleted_line*, a string) and its index in the text (*deleted_line_index*, an integer). This is the information needed to undo the command should this be required later on, or to redo it.

A command object

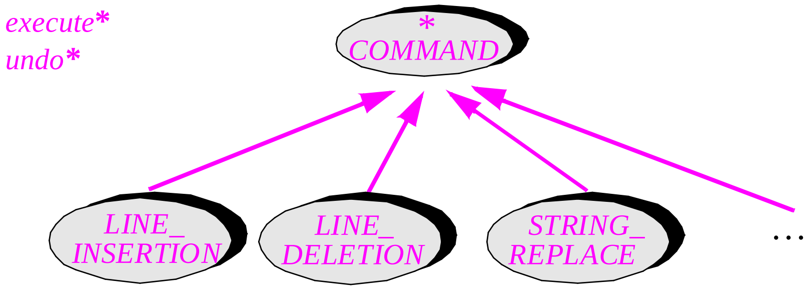


See “Requirements on the solution”, page 697.

The exact attributes — such as *deleted_line* and *deleted_line_index* here — will differ for each command class, but they should always be sufficient to support the local variants of *execute* and *undo*. Such objects, conceptually describing the difference between the states that precede and follow the application of a command, will enable us to satisfy requirement U3 of the earlier list — storing only what is strictly necessary.

The inheritance structure of command classes may look like this:

Command class hierarchy



The graph shown is flat (all proper descendants of *COMMAND* at the same level), but nothing precludes adding more structure by grouping command types into intermediate categories; this will be justified if such categories make sense as abstract data types, that is to say, have specific features.

When defining a notion, it is always important to indicate what it does not cover. Here the concept of command does not include Undo and Redo; for example it would not make sense to undo an Undo (except in the sense of doing a Redo). For this reason the discussion uses the term *operation* for Undo and Redo, reserving *command* for operations which can be undone and redone, such as line insertion. There is no need for a class covering the notion of operation, since non-command operations such as Undo have only one relevant feature, their ability to be executed.

“The nouns and the verbs”, page 720.

This is a good example of the limitations of simplistic approaches to “find the objects”, such as the famous “Underline the nouns” idea studied in a later chapter. In the specification of the problem, the nouns *command* and *operation* are equally important; but one gives a fundamental class, the other does not give a class at all. Only the abstract data type perspective — studying abstractions in terms of the applicable operations and their properties — can help us find the classes of our object-oriented systems.

The basic interactive step

To get started we will see how to support one-level undo. The generalization to multi-level undo-redo will come next.

In any interactive system, there must be somewhere, in a module in charge of the communication with users, a passage of the form

```
basic_interactive_step is
        -- Decode and execute one user request.
    do
        "Find out what the user wants us to do next"
        "Do it (if possible)"
    end
```

In a traditionally structured system, such as editor, these operations will be executed as part of a loop, the program's "basic loop":

```
from start until quit_has_been_requested_and_confirmed loop
    basic_interactive_step
end
```

whereas more sophisticated systems may use an event-driven scheme, in which the loop is external to the system proper (being managed by the underlying graphical environment). But in all cases there is a need for something like *basic_interactive_step*.

In light of the abstractions just identified, we can reformulate the body of the procedure as

```
"Get latest user request"
"Decode request"
if "Request is a normal command (not Undo)" then
    "Determine the corresponding command in our system"
    "Execute that command"
elseif "Request is Undo" then
    if "There is a command to be undone" then
        "Undo last command"
    elseif "There is a command to be redone" then
        "Redo last command"
    end
else
    "Report erroneous request"
end
```

This implements the convention suggested earlier that Undo applied just after Undo means Redo. A request to Undo or Redo is ignored if there is nothing to undo or redo. In a simple text editor with a keyboard interface, “Decode request” would analyze the user input, looking for such codes as control-I (for insert line), control-D (for delete line) and so on. With graphical interfaces you have to determine what input the user has entered, such as a choice in a menu, a button clicked in a menu, a key pressed.

Remembering the last command

With the notion of command object we can be more specific about the operations performed by *basic_interactive_step*. We will use an attribute

```
requested: COMMAND
        -- Command requested by interactive user
```

representing the latest command that we have to execute, undo or redo. This enables us to refine the preceding scheme of *basic_interactive_step* into:

```
“Get and decode latest user request”
if “Request is normal command (not Undo)” then
    “Create appropriate command object and attach it to requested ”
        -- requested is created as an instance of some
        -- descendant of COMMAND, such as LINE_DELETION
        -- (This instruction is detailed below.)
    requested.execute ; undoing_mode := False          Dynamic Binding
elseif “request is Undo” and requested /= Void then
    if undoing_mode then
        “This is a Redo; details left to the reader”
    else
        requested.undo ; undoing_mode := True
    end
else
    “Erroneous request: output warning, or do nothing”
end
```

The boolean entity *undoing_mode* determines whether the last operation was an Undo. In this case an immediately following Undo request would mean a Redo, although the straightforward details have been left to the reader; we will see the full details of Redo implementation in the more interesting case of a multi-level mechanism.

Exercise E21.2, page 716.

The information stored before each command execution is an instance of some descendant of *COMMAND* such as *LINE_DELETION*. This means that, as announced, the solution satisfies the property labeled U3 in the list of requirements: what we store for each command is the difference between the new state and the previous one, not the full state.

See “Requirements on the solution”, page 697.

The key to this solution — and its refinements in the rest of this chapter — is polymorphism and dynamic binding. Attribute *requested* is polymorphic: declared of type *COMMAND*, it will become attached to objects of one of its effective descendant types such as *LINE_INSERTION*. The calls *requested*.*execute* and *requested*.*undo* only make sense because of dynamic binding: the feature they trigger must be the version redefined for the corresponding command class, executing or undoing a *LINE_INSERTION*, a *LINE_DELETION* or a command of any other type as determined by the object to which *requested* happens to be attached at the time of the call.

The system's actions

No part of the structure seen so far is application-specific. The actual operations of the application, based on its specific object structures — for example the structures representing the current text in a text editor — are elsewhere; how do we make the connection?

The answer relies on the *execute* and *undo* procedures of the command classes, which must call application-specific features. For example procedure *execute* of class *LINE_DELETION* must have access to the editor-specific classes to call features that will yield the text of the current line, give its position in the text, and remove it.

As a result there is a clear separation between the user interaction parts of a system, largely application-independent, and the application-specific parts, closer to the model of each application's conceptual model — be it text processing, CAD-CAM or anything else. The first component, especially when generalized to a history mechanism as explained next, will be widely reusable between various application domains.

How to create a command object

After decoding a request, the system must create the corresponding command object. The instruction appeared abstractly as "Create appropriate command object and attach it to *requested*"; we may express it more precisely, using creation instructions, as

```
if "Request is LINE INSERTION" then
        ! LINE_INSERTION ! requested.make (input_text, cursor_index)
elseif "Request is LINE DELETION" then
        ! LINE_DELETION ! requested.make (current_line, line_index)
elseif
        ...
```

"Polymorphic creation", page 479.

This uses the ! *SOME_TYPE* ! *x* … form of the creation instruction, which creates an object of type *SOME_TYPE* and attaches it to *x*; remember that *SOME_TYPE* must conform to the type declared for *x*, as is the case here since *requested* is of type *COMMAND* and all the command classes are descendants of *COMMAND*.

If each command type uses a **unique** integer or character code, a slightly simpler form relies on an **inspect**:

```
inspect
    request_code
when Line_insertion then
    ! LINE_INSERTION ! requested.make (input_text, cursor_position)
etc.
```

Both forms are multiple-branch choices, but they do not violate the Single Choice principle: as was pointed out in the discussion of that principle, if a system provides a number of alternatives some part of it *must* know the complete list of alternatives. The above extract, in either variant, is that point of single choice. What the principle precludes is spreading out such knowledge over many modules. Here, no other part of the system needs access to the list of commands; every command class deals with just one kind of command.

“Single Choice”, page 61.

It is in fact possible to obtain a more elegant structure and get rid of the multi-branch choice totally; we will see this at the end of presentation.

“Precomputing command objects”, page 708.

21.3 MULTI-LEVEL UNDO-REDO

Supporting an arbitrary depth of undoing, with the attendant redoing, is a straightforward extension of the preceding scheme.

The history list

What has constrained us to a single level of undoing was the use of just one object, the last created instance of *COMMAND* available through *requested*, as the only record of previously executed commands.

In fact we create as many objects as the user executes commands. But because the software only has one command object reference, *requested*, always attached to the last command, every command object becomes unreachable as soon as the user executes a new command. It is part of the elegance and simplicity of a good O-O environment that we do not need to worry about such older command objects: the garbage collector will take care of reclaiming the memory they occupy. It would be a mistake to try to reclaim the command objects ourselves, since they may all be of different shapes and sizes.

See chapter 9 on garbage collection.

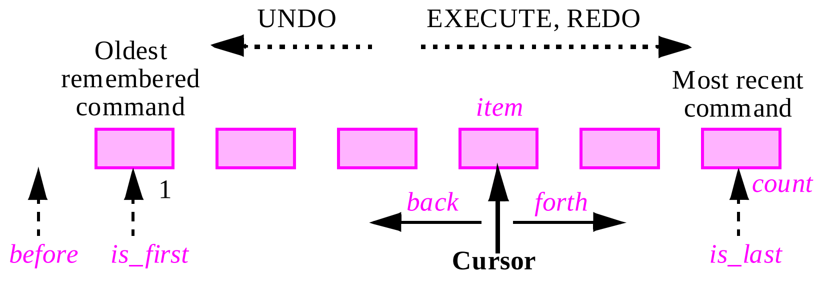
To provide more depth of undoing we need to replace the single command *requested* by a list of recently executed commands, the history list:

```
history: SOME_LIST [COMMAND]
```

SOME_LIST is not a real class name; in true object-oriented, abstract data type style we will examine what features and properties we need from *SOME_LIST* and draw the conclusion as to what list class (from the Base library) we can use. The principal operations we need are straightforward and well known from previous discussions:

- *put* to insert an element at the end (the only place where we will need insertions). By convention, *put* will position the list cursor on the element just inserted.
- *empty* to find out whether the list is empty.

A history list



- *before*, *is_first* and *is_last* to answer questions about the cursor position.
- *back* to move the cursor back one position and *forth* to advance it one position.
- *item* to access the element at cursor position, if any; this feature has the precondition (**not** *empty*) **and** (**not** *before*), which we can express as a query *on_item*.

In the absence of undoing, the cursor will always be (except for an empty list) on the last element, making *is_last* true. If the user starts undoing, the cursor will move backward in the list (all the way to *before* if he undoes every remembered command); if he starts redoing, the cursor will move forward.

Skip is the subject of exercise E21.4, page 716.

The figure shows the cursor on an element other than the last; this means the user has just executed one or more Undo, possibly interleaved with some Redo, although the number of Undo must always be at least as much as the number of Redo (it is greater by two in the state captured in the figure). If at that stage the user selects a normal command — neither Undo nor Redo —, the corresponding object must be inserted immediately to the right of the cursor element. The remaining elements on the right are lost, since Redo would not make sense in that case; this is the same situation that caused us at the beginning of this chapter to relegate the notion of Skip operation to an exercise. As a consequence we need one more feature in *SOME_LIST*: procedure *remove_all_right*, which deletes all elements to the right of the cursor.

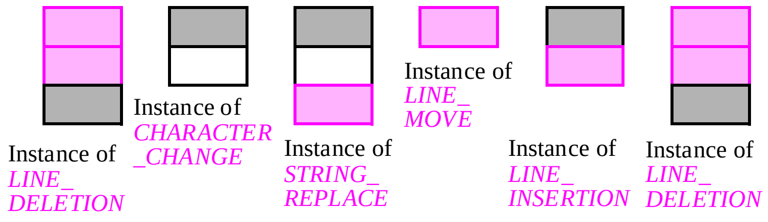
An Undo is possible if and only if the cursor is on an element, as stated by *on_item*. A Redo is possible if and only if there has been at least one non-overridden Undo, that is to say, (**not** *empty*) **and** (**not** *is_last*), which we may express through a query *not_last*.

Implementing Undo

With the history list, it is easy to implement Undo:

```
if on_item then
    history.item.undo
    history.back
else
    message ("Nothing to undo")
end
```

See once again how dynamic binding is essential. The *history* list is a polymorphic data structure:



A history list with its various command objects

As the cursor moves left, each successive value of *history*.*item* may be attached to an object of any of the available command types; in each case, dynamic binding ensures that *history*.*item*.*undo* automatically selects the appropriate version of *undo*.

Implementing Redo

Redo is similar:

```
if not_last then
    history.forth
    history.item.redo
else
    message ("Nothing to redo")
end
```

This assumes a new procedure, *redo*, in class *COMMAND*. So far we had taken for granted that *redo* is the same thing as *execute*, and indeed in most cases it will be; but for some commands re-executing after an undo might be slightly different from executing from scratch. The best way to handle such situations — providing enough flexibility, without sacrificing convenience for the common cases — is to provide the default behavior in class *COMMAND*:

```
redo is
            -- Re-execute command that has been undone
            -- by default, the same thing as executing it.
    do
        execute
    end
```

This makes *COMMAND* a behavior class: along with deferred *execute* and *undo*, it has an effective procedure *redo* which defines a behavior based, by default, on the other two. Most descendants will keep this default, but some of them may redefine *redo* to account for special cases.

"Don't call us, we'll call you", page 505.

Executing a normal command

If a user operation is neither Undo nor Redo, it is a normal command identified by a reference that we may still call *requested*. In this case we must execute the command, but we must also insert it into the history list; we should also, as noted, forget any item to the right of the cursor. So the sequence of instructions is:

```
if not is_last then remove_all_right end
history.put (requested)
                        -- Recall that put inserts at the end of the list and moves
                        -- the cursor to the new element.
requested.execute
```

With this we have seen all the essential elements of the solution. The rest of this chapter discusses a few implementation-related topics and draws the methodological lessons from the example.

21.4 IMPLEMENTATION ASPECTS

Let us examine a few details that help obtain the best possible implementation.

Command arguments

Some commands will need arguments. For example a *LINE_INSERTION* needs to know the text of the line to be inserted.

A simple solution is to add to *COMMAND* an attribute and a procedure:

```
argument: ANY
set_argument (a: like argument) is
    do argument := a end
```

Then any command class can redefine *argument* to the proper type. To handle multiple arguments, it suffices to choose an array or list type. This was the technique assumed above when we passed various arguments to the creation procedures of command classes.

This technique is appropriate for all simple applications. Note, however, that the *COMMAND* class in ISE’s libraries uses a different technique, slightly more complicated but more flexible: there is no *argument* attribute, but procedure *execute* takes an argument (in the usual sense of argument to a routine), representing the command argument:

```
execute (command_argument: ANY) is …
```

The reason is that it is often convenient, in a graphical system, to let different instances of the same command type share the same argument; by removing the attribute we can reuse the same command object in many different contexts, avoiding the creation of a new command object each time a user requests a command.

The small complication is that the elements of the history list are no longer instances of *COMMAND*; they must instead be instances of a class *COMMAND_INSTANCE* with attributes

```
command_type: COMMAND
argument: ANY
```

For a significant system, the gain in space and time is worth this complication, since you will create one command object per command type, rather than one per command execution. This technique is recommended for production applications. You will only need to change a few details in the preceding class extracts.

Exercise E21.4, page 716.

Precomputing command objects

Before executing a command we must obtain, and in some cases create, the corresponding command object. The instruction was abstractly written as “Create appropriate command object and attach it to *requested*” and the first implementation draft was

Page 704.

```
inspect
        request_code
when Line_insertion then
        ! LINE_INSERTION ! requested.make (…)
etc. (one branch for each command type)
```

As pointed out, this instruction does *not* violate the Single Choice principle: it is in fact the point of single choice — the only place in the entire system that knows what set of commands is supported. But we have by now developed a healthy loathing for **if** or **inspect** instructions with many branches, so even if this one appears inevitable at first let us see if perhaps we could get rid of it anyway.

We can — and the design pattern, which may be called **precomputing a polymorphic instance set**, is of wide applicability.

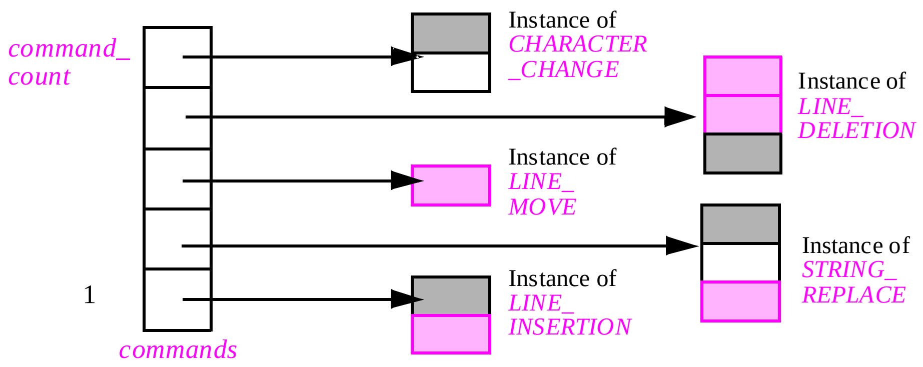
The idea is simply to create once and for all a polymorphic data structure containing one instance of each variant; then when we need a new object we simply obtain it from the corresponding entry in the structure.

Although several data structures would be possible for such as a list, it is most convenient to use an *ARRAY* [*COMMAND*], allowing us to identify each command type with an integer between 1 and *command_count*, the number of command types. We declare

```
commands: ARRAY [COMMAND]
```

and initialize its elements in such a way that the *i*-th element ($1 <= i <= n$) refers to an instance of the descendant class of *COMMAND* corresponding to code *i*; for example, we create an instance of *LINE_DELETION*, associate it with the first element of the array (assuming line deletion has code 1), and so on.

The array of command templates


command_
count
1
commands
Instance of CHARACTER_CHANGE
Instance of LINE_DELETION
Instance of LINE_MOVE
Instance of STRING_REPLACE
Instance of LINE_INSERTION

A similar technique can be applied to the polymorphic array *associated_state* used in the O-O solution to the last chapter's problem (panel-driven applications).

The array *commands* is another example of the power of polymorphic data structures. Its initialization is trivial:

```
!! commands.make (1, command_count)

! LINE_INSERTION ! requested.make; commands.put (requested, 1)
! STRING_REPLACE ! requested.make; commands.put (requested, 2)
… And so on for each command type …
```

Note that with this approach the creation procedures of the various command classes should not have any arguments; if a command class has attributes, they should be set separately later on through specific procedures, as in *li*.*make* (*input_text*, *cursor_position*) where *li* is of type *LINE_INSERTION*.

Then there is no more need for any **if** or **inspect** multi-branch instruction. The above initialization serves as the point of single choice; you can now write the operation "Create appropriate command object and attach it to *requested*" as

```
requested := clone (commands @ code)
```

where *code* is the code of the last command. (Since each command type now has a code, corresponding to its index in the array, the basic user interface operation written earlier as "Decode request" analyzes the user's request and determines the corresponding code.)

The figure showing a history list example was on page 706.

The assignment to *requested* uses a *clone* of the command template from the array, so that you can have more than one instance of the same command type in the history list (as in the earlier example, where the history includes two *LINE_DELETION* objects).

If, however, you use the suggested technique of completely separating the command arguments from the command objects (so that the history list contains instances of *COMMAND_INSTANCE* rather than *COMMAND*), then the clone is not necessary any more, and you can go on using references to the original objects from the array, with just:

```
requested := commands @ code
```

In very long sessions the savings can be significant.

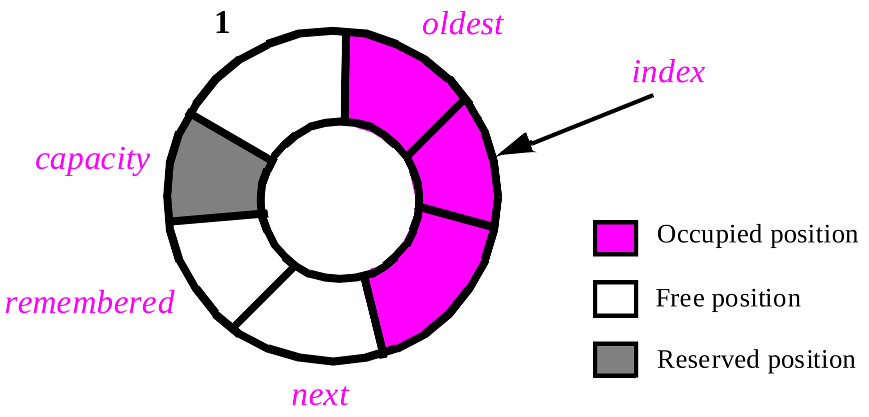
A representation for the history list

For the history list a type *SOME_LIST* was posited, with features *put*, *empty*, *before*, *is_first*, *is_last*, *back*, *forth*, *item* and *remove_all_right*. (There is also *on_item*, expressed in terms of *empty* and *before*, and *not_last*, expressed in terms of *empty* and *is_last*.)

Many of the classes in the Base libraries can be used to implement *SOME_LIST*; for example we could rely on *TWO_WAY_LIST* or one of the descendants of the deferred class *CIRCULAR_LIST*. To obtain a stand-alone solution let us devise an ad hoc class *BOUNDED_LIST*. Unlike a linked implementation such as *TWO_WAY_LIST*, this one will rely on an array, so it keeps only a bounded number of commands in the history. Let *remembered* be the maximum number of remembered commands. If you use this facility for a system to build, remember (if only to avoid receiving an angry letter from me should I ever become a user) to make this maximum user-settable, both during the session and in a permanent user profile consulted at the beginning of each session; and choose a default that is not too small, for example 20.

BOUNDED_LIST can use an array, managed circularly to enable reusing earlier positions as the number of commands goes beyond *remembered*. With this technique, common for representing bounded queues (it will show up again for bounded buffers in the discussion of concurrency), we can picture the array twisted into a kind of doughnut:

See “A buffer is a separate queue”, page 990.



Bounded circular list implemented by an array

The size *capacity* of the array is *remembered + 1*; this convention means setting aside one of the positions (the last, at index *capacity*) and is necessary if we want to be able to distinguish between an empty list and a full list (see below). The occupied positions are marked by two integer attributes: *oldest* is the position of the oldest remembered command, and *next* is the first free position (the one at which the next command will be inserted). The integer attribute *index* indicates the current cursor position.

An alternative to reserving a free position is to keep track of count, the number of elements, in an attribute. See “Unconstrained genericity”, page 1181 for yet another variant.

Here is the implementation of the various features. For *put* (*c*), inserting command *c* at the end of the list, we execute

```
representation.put (x, next);       -- where representation is the name of the array
next := (next \\ remembered) + 1
index := next
```

where \\ is the integer remainder operation. The value of *empty* is true if and only if *next* = *oldest*; that of *is_first*, if and only if *index* = *oldest*; and that of *before* if and only if (*index* \\ *remembered*) + 1 = *oldest*. The body of *forth* is

```
index := (index \\ remembered) + 1
```

and the body of *back* is

```
index := ((index + remembered – 2) \\ remembered) + 1
```

The +*remembered* term is mathematically redundant, but is included because of the lack of universal conventions as to the computer meaning of remainder operations for negative operands.

The query *item* giving the element at cursor position returns *representation* @ *index*, the array element at index *index*. Finally, the procedure *remove_all_right*, removing all elements to the right of the cursor position, is simply implemented as

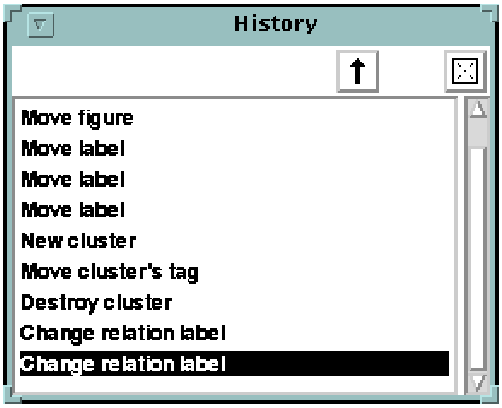
```
next := (index \\ remembered) + 1
```

21.5 A USER INTERFACE FOR UNDOING AND REDOING

Here is part of a possible user interface support for the undo-redo mechanism. It is taken from ISE's Case analysis and design workbench, but several of our other products use the same scheme.

Although keyboard shortcuts are available for Undo and Redo, the complete mechanism involves bringing up a **history window** (by clicking on a button in the interface, or selecting an item in the Tools menu). The history window is the exact user-visible equivalent of the *history* list as it exists inside the software. Once it is up, it will be regularly updated as you execute commands and other operations. In the absence of any undoing, it will look like this:

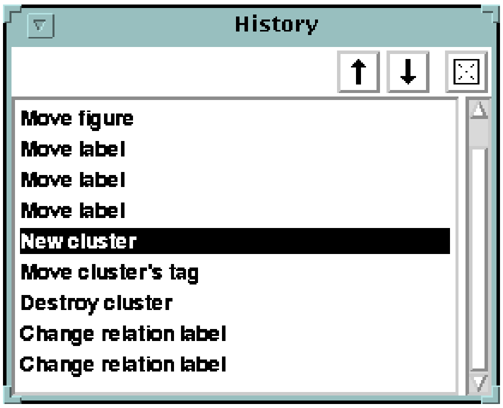
A history window, before any undoing


History

Move figure
Move label
Move label
Move label
New cluster
Move cluster's tag
Destroy cluster
Change relation label
Change relation label

This shows the list of recent commands. As you execute new commands, they will appear at the end of the list. The currently active command (the one at cursor position) is highlighted in inverse video, like **change relation label** on the last figure.

To undo the active command, you can click on the up arrow button ↑ or use the keyboard shortcut (such as ALT-U). The cursor moves up (back) in the list; after a few such Undo, the window would look like this:



Move figure
Move label
Move label
Move label
New cluster
Move cluster's tag
Destroy cluster
Change relation label
Change relation label

A history window, in the middle of an undo-redo process

As you know, this internally means that the software has been performing a few calls to *back*. At this stage you have a choice between several possibilities:

- You can perform more Undo operations by clicking on the up arrow button; the highlighting moves to the previous line.
- You can perform one or more Redo by clicking on the down arrow ↓ or using the equivalent keyboard shortcut; the highlighting goes to the next line, internally performing calls to *forth*.
- You can execute a normal command. As we have seen, this will remove from the history any commands that have been undone but not redone, internally performing a *remove_all_right*; in the interface, all the commands below the currently highlighted one disappear.

21.6 DISCUSSION

The design pattern presented in this chapter has an important practical role, as it will enable you to write significantly better interactive systems at little extra effort. It also brings an interesting theoretical contribution, by illuminating some aspects of object-oriented methodology worth exploring further.

The role of implementation

A striking property of the example user interface presented in the last section is that it was *directly deduced* from the implementation: we took the internal, developer-relevant notion of history list and translated it into an external, user-relevant history window, with the attendant user interaction mechanism.

One may always imagine that someone could have devised the external view first, or at any rate independently from the implementation. But this is not the way it happened, either in this presentation or in history of our products' development.

Instituting such a relation between a system's functionality and its implementation goes against all that traditional software engineering methodology has taught. We have been told to deduce the implementation from the specification, not the reverse! Techniques of "iterative development" and "spiral lifecycle" change little to this fundamental rule that implementation is slave to prior concept, and that the software developers must do what the "users" (meaning, the customers, usually non-technical) tell them. Here we are violating every taboo by asserting that the *implementation* can tell us *what the system should be doing* in the first place. In earlier times questioning such time-honored definitions of what depends on what could have led one to the stake.

The legitimate emphasis on involving customers — meant to avoid the all too common horror stories of systems that do not do what their users need — has unfortunately led to downplaying the software developers' contribution, whose importance extends to the most external and application-related aspects. It is naïve to believe, for example, that customers will suggest the right interface facilities. Sometimes they will, but often they reason on the basis of the systems they know, and they will not see all the issues involved. That is understandable: they have their own jobs to do, and their own areas of expertise; getting everything right in a software system is not their responsibility. Some of the worst interactive interfaces in the world were designed with *too much* user influence. Where users are truly irreplaceable is for negative comments: they will see practical flaws in an idea which at first seems attractive to the developers. Such criticism must always be heeded. Users can make brilliant positive suggestions too, but do not depend on it. And once in a while, a developer's suggestion will seduce the users — possibly after a number of iterations taking their criticism into account — even though it draws its origin from a seemingly humble implementation technique, such as the history list.

More on seamlessness and reversibility in chapter 28.

This equalization of traditional relationships is one of the distinctive contributions of object technology. By making the development process seamless and reversible, we allow a great implementation idea to influence the specification. Instead of a one-way flow from analysis to design and "coding", we have a continuous process with feedback loops throughout. This assumes, of course, that implementation is no longer viewed as the messy, low-level component of system construction; its results, developed with the techniques described throughout this book, can and should be as clear, elegant and abstract as anything one can produce in the most implementation-abhorrent forms of traditional analysis and design.

Small classes

The design described in this chapter may, for a typical interactive system, involve a significant number of relatively small classes: one for each type of command. There is no reason, however, to be concerned about the effect on system size and complexity since the inheritance structure on these classes will remain simple, although it does not have to be as flat as the one sketched in this chapter. (You may want to group commands into categories.)

In a systematic O-O approach, similar questions arise whenever you have to introduce classes representing actions. Although some object-oriented languages make it possible to pass routines as arguments to other routines, such a facility contradicts the basic idea of the method — that a function (action, routine) never exists by itself but is always *relative to a certain data abstraction*. So instead of passing an operation we should pass an object equipped, through a routine of its generating class, with that operation, as with an instance of *COMMAND* equipped with the *execute* operation.

Sometimes the need to write a wrapper class seems artificial, especially to people used to passing routines around as arguments. But every time I have seen such a class legitimately being introduced, originally for the sole purpose (it was thought) of encapsulating an operation, it turned out to reveal a useful data abstraction, as evidenced by the later addition of other features beyond the one that served as the original incentive. Class *COMMAND* does not fall into this category, since right from the start it was conceived as a data abstraction, and had two features (*execute* and *undo*). But it is typical of the process, since if you start using commands seriously you will soon realize the need for even more features such as:

- *argument*: *ANY* to represent the command argument (as in one of the versions that we have encountered).
- *help*: *STRING*, to provide on-line help associated with each command.
- Logging and statistical features, to keep track of how often each command type is used.

Another example, drawn from the domain of numerical software, is more representative of situations where the introduction of a class may seem artificial at first, because the object-oriented designer will pass an object where a traditional approach would have passed a routine as argument. In performing scientific computation you will often need integration mechanisms, to which you give a mathematical function *f* to compute its integral on a certain interval. The traditional technique is to represent *f* as a routine, but in object-oriented design we recognize that “Integrable function” is an important abstraction, with many possible features. For someone coming from the functional world of C, Fortran and top-down design, the need to provide a class may at first appear to be a kind of programming trick: not finding in the language manual a way to pass a routine as argument, he asks his colleagues how to achieve this effect, and is told that he must write a class with the corresponding feature, then pass objects (instances of that class) rather than the feature itself.

He may at first accept this technique — perhaps grudgingly — as one of those quirks that programming languages impose on their users, as when you want a boolean variable in C and have to declare it of type integer, with 0 for false and 1 for true. But then as he continues his design he will realize that the technique was not a hack, simply the proper application of object-oriented principles: *INTEGRABLE_FUNCTION* is indeed one of the major abstractions of his problem domain, and soon new, relevant features (beyond the original one *item* (*a*: *REAL*): *REAL*, giving the value of the function at point *a*) will start piling up.

What was thought to be a trick turns out to yield a major component of the design.

21.7 BIBLIOGRAPHICAL NOTES

The undo-redo mechanism described in this chapter was present in the structural document constructor *Cépage* developed by Jean-Marc Nerson and the author in1982 [M 1984], and has been integrated into many of ISE's interactive tools (including ArchiText [ISE 1996], the successor to Cépage).

In [Cox 1986].

In a position paper for a panel at the first OOPSLA conference in 1986, Larry Tesler cites a mechanism based on the same ideas, part of Apple's *MacApp* interactive framework.

[Dubois 1997] explains in detail how to apply object-oriented concepts to the design of numerical software, with abstractions such as "Integrable function" (as mentioned in the last section), and describes in detail a complete object-oriented numerical library.

EXERCISES

E21.1 Putting together a small interactive system (programming project)

This small programming project is an excellent way to test your understanding of the topics of this chapter — and more generally of how to build a small system making full use of object-oriented techniques.

Write a line-oriented editor supporting the following operations:

- **p**: Print text entered so far.
- ↓: move cursor to next line if any. (Use the code **l**, for low, if that is more convenient.)
- ↑: move cursor to previous line if any. (Use **h**, for high, if that is more convenient.)
- **i**: insert a new line after cursor position.
- **d**: delete line at cursor position.
- **u**: Undo last operation if not Undo; if it was Undo, redo undone command.

You may add more commands, or choose a more attractive user interface, but in all cases you should produce a complete, workable system. (You may also apply right from the start the improvement described in the next exercise.)

E21.2 Multi-level Redo

Complete the previous exercise’s one-level scheme by redefining the meaning of **u** as

- **u**: Undo last operation other than Undo and Redo.

and adding

- **r**: Redo last undone command (when applicable).

E21.3 Undo-redo in Pascal

Explain how to obtain a solution imitating the undo-redo technique of this chapter in non-O-O languages such as Pascal, Ada (using record types with variants) or C (using structure and union types). Compare with the object-oriented solution.

E21.4 Undo, Skip and Redo

Bearing in mind the issues raised early in the discussion, study how to extend the mechanism developed in this chapter so that it will support Undo, Skip and Redo, as well as making it possible to redo an undone command that has been followed by a normal command.

See “Practical issues”, page 696 and “The history list”, page 704.

Discuss the effect on both the user interface and the implementation.

E21.5 Saving on command objects

Adapt all the class extracts of this chapter to treat command arguments separately from commands (adding a routine argument to *execute*) and create only one command object per command type.

See “Command arguments”, page 707.

If you have done the preceding exercise, apply this technique to its solution.

E21.6 Composite commands

For some systems it may be useful to introduce a notion of composite command, describing commands whose execution involves executing a number of other commands. Write the corresponding class *COMPOSITE_COMMAND*, an heir of *COMMAND*, making sure that composite commands can be undone, and that a component of a composite command may itself be composite.

See “Composite figures”, page 527.

Hint: use the multiple inheritance scheme presented for composite figures.

E21.7 Non-undoable commands

A system may include commands that are not undoable, either by nature ("Fire the missiles") or for pragmatic reasons (when there is too much information to remember). Refine the solution of this chapter so that it will account for non-undoable commands. (**Hint**: introduce heirs *UNDOABLE* and *NON_UNDOABLE* to class *COMMAND*.) Study carefully the effect on the algorithms presented, and on the user interface, in particular for an interface using the history windows as presented at the end of the chapter.

E21.8 A command library (design and implementation project)

Write a general-purpose command library, meant to be used by an arbitrary interactive system and supporting an unlimited undo-redo mechanism. The library should integrate the facilities discussed in the last three exercises: separating commands from arguments; composite commands; non-undoable commands. (Integrating an "Undo, Skip and Redo" facility is optional.) Illustrate the applicability of your library by building three demonstration systems of widely different natures, such as a text editor, a graphics system and a training tool.

E21.9 A history mechanism

A useful feature to include in a command-oriented interactive tool is a history mechanism which remembers the last commands executed, and allows the user to re-execute a previous command, possibly modified, using simple mnemonics. Under Unix, for example, you may direct the C-shell (a command language) to remember the last few executed commands; then you may type *!–2* to mean "re-execute the next-to-last command", or *^yes^no^* to mean "re-execute the last command, replacing the characters *yes* in the command text by *no*". Other environments offer similar facilities.

History mechanisms, when they exist, are built in an ad hoc fashion. On Unix, many interactive tools running under the C-shell, such as the Vi editor or various debuggers, would greatly benefit from such a mechanism but do not offer one. This is all the more regrettable that the same concept of command history and the same associated facilities are useful for any interactive tool independently of the functions it performs — command language, editor, debugger.

Design a class implementing a general-purpose history mechanism, in such a way that any interactive tool needing such a mechanism will obtain it by simply inheriting from that class. (Note that multiple inheritance is essential here.)

Discuss the extension of this mechanism to a general *USER_INTERFACE* class.

E21.10 Testing environment

Proper testing of a software component, for example a class, requires a number of facilities to prepare the test, input test data, run the test, record the results, compare them to expected results etc. Define a general *TEST* class that defines an appropriate testing environment and may be inherited by any class in need of being tested. (Note again the importance of multiple inheritance.)

E21.11 Integrable functions

(For readers familiar with the basics of numerical analysis.) Write a set of classes for integrating real functions of a real variable over arbitrary intervals. They should include a class *INTEGRABLE_FUNCTION*, as well as a deferred class *INTEGRATOR* to describe integration methods, with proper descendants such as *RATIONAL_FIXED_INTEGRATOR*.

See "Small classes", page 714. For a full-fledged solution see [Dubois 1997].