Branching Undo/Redo Mechanism Based on Variadic Parameter Command Pattern

Yingda Li, Jianzhuang Lu and Xiaowen Chen

ABSTRACT

Traditional linear undo/redo mechanism cannot backtrack to some special history nodes. For example, the users undo to a previous state and then submit a new operation. In this case, the undone history node cannot be recovered anymore. To solve this problem, a branching undo/redo mechanism that can retain all the undone history nodes is proposed. Its undo/redo realization based on command pattern is detailed. Besides, a new universal variadic parameter command pattern is proposed to overcome the disadvantages of classic command pattern. The variadic parameter command pattern can be adapt to different command receivers and different receiver actions with different action arguments.

INTRODUCTION

Background

Undo/redo mechanism is of great importance in editing software such as Word and Photoshop, which gives the users the option to undo the mistaking or undesirable operations. The users will be willing to have different tries when they can undo or redo any operations and this can give them full play to their creativity [1]. Undo/redo mechanism is widely used now, however, most of them are liner undo/redo. Undo and redo can only be executed along a line in one-dimensional space. As shown in figure 1, the state of an edit moves from A to B, then it is undone back to A and edited to C.

Li Yingda, Lu Jianzhuang, Chen Xiaowen
College of Computer, National University of Defense Technology, Kaifu, Changsha, China.
In this case, only state A and state C are available with linear undo/redo. The linear undo/redo mechanism can't achieve state B, so we put forward branching undo/redo mechanism. As shown in figure 2, branching undo/redo mechanism retains the states that be overwritten by redo operations and allow the users to backtrack to these states. This mechanism is especially important in graphic editing software because it provides the designers with convenience to make comparison and choice.

**Review**

Undo/redo mechanism is widely used in editing software and there are many methods to realize it. A simple way is to make a copy of the program’s objects and push them in a stack before the state has changed. These objects could be popped out to overwrite the current objects when undo is in need [2]. This method is very easy but all of the objects should be backed up every time before a new operation, which will bring a large number of redundant data and use a lot of memory.

With the help of software design patterns, undo/redo mechanism can be realized by making use of command pattern [3]. Command pattern encapsulates operation requests in command objects that inherit the same command interface and parameterizes operation requests with command objects. Undo method could be added to the command interface to draw back a command. Undo is an inverse operation of executing a command and it uses the information backed up before execution to roll back to the earlier states. To support multilevel undo/redo, command objects can be saved in a history list in order. Multilevel undo is realized by calling undo function of each command object one by one along the opposite direction of the history list. This is a linear undo/redo mechanism.

Thomas Berlage has put up selective undo in his classical research [4]. Selective undo can undo a much earlier isolated command but does not affect the commands in between. This research mentions history tree that is used to extend the linear undo. It also puts up the inverse model, which adds another inverse command at the end of history list other than undo the command directly. Regional undo is put up in another research [5], which allows the user to select a specific editing region and undo the last command in this region. Selective undo/redo is used in painting software in recent years [6], and the advantages of script mode [7] are pointed out. Script mode tries to
guarantee that the result of undo is as if the previous command had never been executed.

Branching undo/redo mechanism has never attracted attention in this area. Even though Thomas Berlage mentioned the concept about history tree, his research was focused on inverse mode and selective undo. The data structure of history tree, the separation and transformation of history tree’s trunk line and branch line and the cutting of size-limited history tree have not been analyzed.

**DESIGN OF UNDO/REDO MECHANISM**

**Branching Structure**

In general liner undo/redo mechanism, when the state is undid to a previous state and then a new operation is executed, the undone node will be deleted from the history list and a new operation node will overwrite it. The user will can’t backtrack to the undone node even if it may be a better one.

If a new history branch could be created when a new operation is executed after undo, the undone node will be retained in the old branch. In this case, all history node could be backtracked to. History branch could be created on any history node by undo operation. Further on, history tree formed. As shown in Figure 3, history tree has trunk line and branch line. The line that contains the latest operation node is trunk line and the other lines are branch lines. Trunk line and branch line can interconvert, which keeps the latest operation node is always on the trunk line.

With the increase in number of operation nodes, the history tree will need more and more memory, so there should be a limit to the size of the history tree. When the number of operation nodes exceed the limit, the history tree should be cut along the trunk line from the root node. If there is a branch line on the node to be cut, the whole branch line should be deleted.

**Node Skip and Branch Skip**

Along the trunk line, the current state could skip from one node to another. Skip along the opposite direction to undo and skip along the positive direction to redo. The essence of node skip is executing the node command object’s undo function or redo function, which is similar to the liner undo/redo mechanism.

Branch skip supports the skip from one line to another line and backtracking to arbitrary node on the history tree, which can reproduce the state that has been undone and overwritten by a new operation. According to the definition of trunk line and branch line, the branch skip is always from trunk line to branch line. At the same time, the target branch line converts to trunk line and the former trunk line converts to
branch line. Branch skip can be separated into two types, single-step skip and direct skip.

The process of single-step skip is to backtrack to a history node that contains a branch one by one with the help of undo, then choose the target branch line by redo and continue to redo until arrive the target node. The user operation and the program internal of single-step skip are identical.

The operation of direct skip is to skip from current node to the target node on the target branch line directly. The program internal of direct skip have three steps: (1) Backtrack from the target node along the target branch, until the trunk node that contains the target branch and mark this node. (2) Execute undo function from the current node along the trunk line sequentially until the marked node. (3) Select the target branch and execute redo function until the target node.

REALIZATION OF UNDO/REDO BASED ON COMMAND PATTERN

Classic Command Pattern and It’s Disadvantages

Command pattern [3] has been widely used in software design, it encapsulates operation request in a command object which can help to queue the operation
requests, record operation log and undo or redo the operations. The structure of command pattern is shown as Figure 4.

- Command provides public interfaces for all concrete commands.
- ConcreteCommand contains the receiver of this command and the receiver’s specific action that will be called in this command.
- Client will create a ConcreteCommand object and set it to the invoker.
- Invoker holds a command object. The command’s interface function will be called when executing this command.
- Receiver contains the action functions that define the detail of operation requests.

However, each operation requests a corresponding command class, so the program may need a large number of command classes, which causes problems for the utilization of command pattern. In classic command pattern, a class template could be used to parameterize the command receiver for the simple commands that don’t need arguments and can’t be undid. However, this solution is only applicable for simple command because the complex commands need to not only maintain the receiver but also log the arguments of functions, and record other information for undo sometimes [3].

**Variadic Parameter Command Pattern**

In program design, the changeable part should be separated out and encapsulated which is benefit for the subsequent change and extension [8]. The changeable part among different commands includes command receiver, receiver action and action arguments. These elements could be separated out as command’s parameters that can be configured.

In C++ program language, we can define a command class template that use receiver as template parameter. This class also has a function pointer that points to the receiver’s action member function. By this way, the parameterized receiver and parameterized action will be bound together.

The receiver and action have been parameterized by command class template. But so far the arguments of action functions still can’t be configured dynamically, which limits the command class’s universal property. The universal property command class requires the action function have not only variadic argument type but also variadic argument count. Variadic argument type is the character of C++ template, and variadic argument count is just supported by C++ 11 [9] [10]. Variadic templates is a new character of C++ 11, which allow the programmer define the class template that have uncertain number of template parameter. And now command class template could be rewritten as Figure 5.

The “typename… Args” in line 8 declares the variadic parameter bag and the virtual function declared and defined with this variadic parameter bag. When the
command is executed, the function execute() will get the variadic parameter bag “arg…” and then pass it to the receiver action. The return of action is passed back to the invoker in the type that set in template parameters finally. The structure of variadic parameter command pattern is shown as Figure 6.

The creation and execution of command is shown as Figure 7. The receiver object r is created in line 1. Command object com is created in line 2. The two arguments in constructor of command object are receiver object and receiver action function. The

![Figure 5. Variadic parameter command pattern.](image1)

parameters of command class template are receiver type, return type of action function, and arguments of action function. The command object is executed with arguments in line 3, the arguments are passed to receiver action finally. In this design pattern, the parameters of command class template can be changed with the receiver action. The type and number of template parameter have no limit, so it has remarkable universal property.

![Figure 6. Structure of variadic parameter command pattern.](image2)
Realization of Undo/Redo with Command Pattern

In the object-oriented software, any operations can be divided into three kind of basic operations including creating an object, destructing an object and modifying an object’s member variables. So any operations can be revoked if this three basic operations could be inversed [11]. The inverse operation of creating an object is destructing this object, the inverse operation of destructing an object is reconstructing this object and the inverse operation of modifying an object’s member variables is recovering to the original value.

All the command objects that support undo/redo implement the interface used to get and set the value of receiver’s member variables. The command object get and back up relevant information about the receiver before the action function is executed. This command will be undid if the back-up information is set back to the receiver and this command will be redid if it is executed again. The branching undo/redo mechanism can be realized by storing the command objects in the history tree.

APPLICATION

The liner undo/redo mechanism need only undo button and redo button, but the branching undo/redo mechanism contains the operation of branch select selection, so it should present all the branches that could be selected and provide the user interfaces for node skip and branch skip. One solution is to present the history tree and history nodes by drop-down menu. As shown in Figure 8, the main menu is used as trunk line, the submenu is used as branch line and the menu items represent history nodes. There will be an arrow symbol on the menu item if this item has a submenu, on another word, this history node has a branch. The submenu will be expanded.
when the mouse moves over on it. Another solution is to present the history tree by
topological graph as shown in Figure 3. This solution is much more intuitive and
easier to use but it needs a specific graphic widget.

Branching undo/redo mechanism can be used in any editing software such as
Photoshop. Photoshop only support liner undo/redo mechanism currently. When the
users already have a design style, then undo and change it into another style, the
previous style will can’t be recovered even if it is better than the current one. Fortunately, the branching undo/redo mechanism support this kind of recovery. The
users could have two designs in different branches and then make a comparison and
selection, after which the design will continue along the selected branch.

COMPARISON AND CONCLUSION

Compared with the liner undo/redo mechanism, branching undo/redo
mechanism retains much more history records. Any state could be backtracked with
in the limitation of history node number. It encourages the users to try different
designs bravely and it is convenient for the users to make comparison and selection
between different designs.

Thomas Berlage has mentioned the concept of history tree in his research about
selective undo/redo mechanism but he didn’t give the detail study. This paper gives
the analysis about the structure of history tree, the separation of trunk line and branch
line, the process of branch skip and the cutting of history tree.

This paper also gives the analysis about the realization of undo/redo mechanism
based on command pattern and gives an innovative development for the classic
command pattern, which makes the command class be universal so that it can be
adapt to different kinds of receivers and different kinds of receiver actions.

REFERENCES

Reusable Object-Oriented Software”, pp. 43-160.
4. Thomas Berlage. 1994. A selective undo mechanism for graphical user interfaces based on
6. Brad A. Myers, Ashley Lai, Tam Minh Le, YoungSeok Yoon, Andrew Faulring, and Joel Brandt.
2015. “Selective Undo Support for Painting Applications”. In Proceedings of the 33rd Annual
USA, 4227-4236.