16

Undo, Scripts and Versions

In 1983 Ben Shneiderman published a paper on the concept of "direct manipulation" [1] where users would stop specifying commands about objects that they could not see but would directly manipulate visual objects on the screen. This was part of the important transition away from command-line interfaces to the graphical user interfaces of today. One of the key claims of direct manipulation is that users would learn primarily by trying manipulations of visual objects rather than by reading extensive manuals. A very important part of learning by trying is perceived safety. A user should be able work in an interface with the confidence that no disastrous thing will happen without warning. The most important feature for such safe exploration is an undo operation that works almost everywhere and in the same way. A second feature is that any operation that cannot be undone must be confirmed by the user before passing the "point of no return".

A common remark by people who are uncomfortable with computers is "I don't want to break it." The fear of permanent loss of information, labor or the good working order of the system is a major impediment to learning new systems. When a user is asked to march into the unknown of a new application they want to know if there are "dragons, snakes, bugs or bears" out there. The answer is usually "yes but undo will kill any that cause problems." That faith in the universal applicability of undo is critical to learning new interfaces.

Undo is also important to efficient use of any application. The existence of an undo means that a user need not plan as carefully before taking an action. If some action leads to a bad place, undo will back out. If there are mistakes caused by working too fast, undo can correct them. Rather than carefully predicting the presence or absence of errors, the user simply speeds ahead and lets the computer show them as a result of actions. If a mistake is made it can be undone. The basic architecture for supporting undo also provides mechanisms for scripting languages and alternative versions of a work product. Scripting allows users to automate repetitive behaviors so that they need not consume user time. Versioning allows groups of people to work together while controlling their shared work product.

UNDO issues

For undo to be effective it must be pervasive. The user must know that every action can be undone not just a selected subset. Having only a subset that are undoable means that the user must consider whether the next action is or is not in the set. This defeats the learning and efficiency advantages of undo. However, if undo is to be pervasive, it poses some serious software architecture problems. A modest application may have hundreds of different actions that will cause changes to the model. Many applications will have thousands and all of them must be reliably undoable whenever the undo action is selected.

We generally view undo as restoring the model to the state it was in before a user action was taken. At first this may seem like a model-oriented problem, however, some actions may make several changes to the model. For example a find and replace operation may do any number of replacement changes but the user thinks about them all as one single event. Undoing them one at a time would not be acceptable. Putting a method on the model to perform find and replace would support placing all undo facilities on the model, but that is an architectural consideration that requires clear thought. If a single user action is composed either by the model or the controller of several model changes, how do all those changes get collected so that they can be undone?

We must also think of the implications of model changes. For example when a user changes a cell in a spreadsheet, that change may cause other cells to recompute. Must we save the changes to all of those cells as part of the undo so that they can be restored or can we save only the original cell’s change and let the recomputation restore the others? In the find and replace command all changes must be saved. In the spreadsheet only the source change must be saved. This requires some careful thought when designing an undo facility.

There is also the question of what things must be undoable by an undo operation and which are naturally transient and do not need an undo. For example, does the movement of a scroll-bar need to be undone? It is a user action, however, for most users the easiest undo is to scroll back to the original position. Do selections need to be undone? The model has not changed but the set of selected objects has changed. Solutions to this issue must answer two questions:

1. Is there a natural undo as part of the user interface? (scroll back to the original position)
2. How much user effort would be lost if undo is not present?

A good example of the second question is found in Adobe Photoshop. In most applications selection is not undoable because it is assumed that the user will simply reselect what is desired and no model information has been lost. In Photoshop selection of a region of pixels can be very laborious (3-10 minutes) with careful attention to image edges required and some manual dexterity and care. Therefore in Photoshop selection is an undoable operation.

There is also the issue of granularity of the undo. When dragging an ellipse across a drawing, the model is modified every time the mouse moves. The user would not be pleased with an undo operation that undid each individual mouse movement. Repeatedly striking control-Z to watch the “ellipse movie” play backwards will not make any user happy. What is generally wanted is that the ellipse will go back to where it was before being dragged. The path along the way was not important even though it included many model changes. In a text box an undo may revert the box contents back to the original text before the user started typing rather than play back the individual character insertions and deletions. In a world processor this is more problematic. A simple undo model will undo every user command, which is every key stroke. This can be very tedious when the user wants to restore an entire paragraph to what it was before. Other word processors use punctuation, timing pauses or cursor selection/movement operations to form boundaries for larger scale undo operations. Granularity of undo requires some careful thought.

Lastly there is the implementation of the undo command itself. When the user strikes control-Z, the application must undo whichever one of its hundreds to thousands of actions the user did last. Obviously a giant `switch` statement that encompasses all possible application commands is not going to scale up to large applications. We need a better architectural solution. We need an undo architecture that the generic undo command can use, but whose details are distributed throughout the code and associated with each individual command to be undone.

There are two main architectures for undo. The first is to store a base checkpoint of the model and then to store a series of change records. This is termed *forward undo* because it starts from a known point and reconstructs the model by making changes moving forward in time. There is also *reverse undo* where only the current model is retained and the change records store enough information to convert the current model into the previous model.

Baseline and forward undo

The forward undo mirrors the patch management strategy found in most source code control systems[2]. These systems store a baseline copy of the model. Each action is then represented by a record of the change that it makes. Starting with the baseline we can apply the changes in order up to whatever point we desire. To undo the last change, we remove it from the change history and rebuild the model from the baseline up to the point just before the change to be undone.

For example a simple text editor could have two commands:

- Delete *startIndex, endIndex*
- Insert *startIndex*, *insertString*

These two commands can be use to represent any changes to a text string. A log of these commands constitutes a *patch* or an *edit history* on the original text string. We can augment these two commands with a “StartAction” command. Before the controller begins modifying the model it places a “StartAction” command in the edit log. Thus if a given controller action makes many modifications to the model the undo can remove all of them up to the last StartAction. This provides for the undo of complex operations in a natural way.

The action log for forward undo is simply a list of model change commands and their parameters. All that is needed is enough information to repeat the calls to the model change methods in the same order. A simple way to implement action logging is to provide the ActionLog object as in figure 16.1.

```
public class ActionLog
{
    public void startAction(); // marks the beginning of a controller action
    public void removeLastAction(); // removes all entries back to the last startAction entry
    public void addCommand(int commandID, string parameters);
        // adds a change command to the log
    public replayCommands(Object model);
        // apply each of the commands to the model in order
    public doCommand(Object model, int commandID, string parameters);
        // this must be overridden in a subclass to implement a switch that selects
        // the correct command action, parses the parameters and calls the method.
}
```

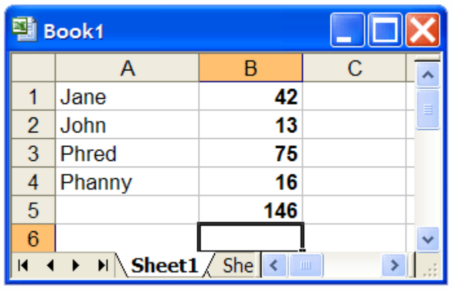
Figure 16.1 – ActionLog for forward undo.

We then take every model change method and add a call to addCommand() to the start to log the change being taken. We also create a subclass of ActionLog and override the doCommand() method with a large switch statement on commandID. This provides the code that will perform the actions by parsing the parameter string and then calling the model method.

This approach is conceptually easy to implement. However, it has three difficulties. The first is that reconstruction of the model through a long series of edits may be inefficient. This is less of a problem with faster machines and the human limits on how many changes human hands can make. The second is that code must be added to every method to correctly log the changes. The third problem is that the doCommand() method becomes the focus of every interactive change in the entire application. This creates serious code management problems.

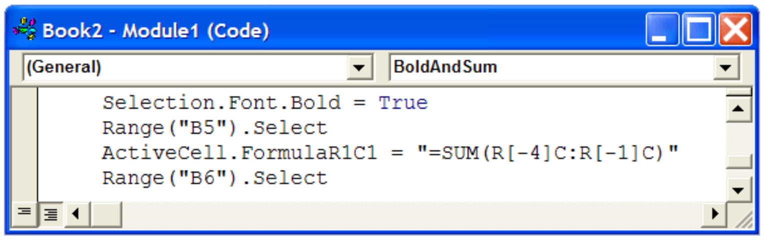
There is a minor variant to this architecture where the addCommand() method is replaced with an doAndAddCommand() method. The doAndAddCommand() method first performs the method and then adds it to the log. The controller then no longer accesses the model directly but through the ActionLog. This removes the logging from the model methods and puts it into the controller. It also ensures that the replay will be identical to the original actions because the model actions were performed through the same mechanism.

If our architecture contains an ActionLog it can also be used for a macro recording system and a scripting system. We first augment ActionLog to include methods to translate string command names to and from the integer commandIDs that we use for efficiency. This allows us to represent each command as *commandName parameterString*. We can create a macro recording system that uses a special ActionLog to collect all of actions between the time that recording is turned on and the time that it is turned off. Replaying those actions using the replayCommands() method will execute the macro. The body of the macro can be saved as a series of command strings for later use. Figure 16.2 shows a small Excel spreadsheet. By turning on macro recording we can record our user interface actions for bolding a column and computing its sum. Figure 16.3 shows the VBA script created as a macro from our actions.



	A	B	C
1	Jane	42	
2	John	13	
3	Phred	75	
4	Phanny	16	
5		146	
6			

Figure 16.2 – Example spreadsheet



```
     Selection.Font.Bold = True
     Range("B5").Select
     ActiveCell.FormulaR1C1 = "=SUM(R[-4]C:R[-1]C)"
     Range("B6").Select
```

Figure 16.3 – Recorded macro

Once we have a mechanism to translate a *command/parameter* string into any action on our model, our interface is now *scriptable*. Once our action/macro system is represented in string form we can edit the macros and scripts. It is now possible to write short programs to augment our interface to do any number of things. A scripting language like our ActionLog is quite simplistic but it is a start. The ability to demonstrate a script in the user interface greatly simplifies the learning of such languages. Microsoft's VBScript and VBA are examples of such tools[3]. The implementation of the TCL[4] scripting language uses a very similar approach.

Input event logging

An alternative to logging model change actions is to log input events. Obviously all model changes are derived from input events and thus a replay of the input events should replay the actions. The attraction of this approach is that there are many fewer input events than there are model actions. There are a few tens of actual input events and even in systems that have hundreds of outside events, the set is fixed and the logging need only be implemented once for all applications. The problem is that the meaning of input events is highly dependent upon the view. Consider for example a web page editor. If the width of the window is changed then the position of the words moves and the meaning of all mouse locations are broken. There are workarounds for these problems but they quickly become more complex than logging the model actions. Input event logging has its attractions but generally is not practical.

Command objects and backup undo

The action log/forward undo model has problems with concentrating all of the interactive command implementations through one doCommand() method. It also has problems with its forward approach of recomputing a model from its baseline. A more efficient approach is for undo to restore the model to what it was before the action was taken. This *backward undo* model requires that we remember enough information to make that restoration. For example when characters are deleted from a document we must remember those characters and where they were in the document so that we can put them back if the undo operation is invoked.

To deal with these issues Bertram Meyer created *command objects*[5]. The approach is based on the Command interface shown in figure 16.4. Using this interface we create a command object class for every model command that we have in our application. Rather than calling a model method, the controller creates the appropriate Command object for that action, invokes doIt() on the object to have the command performed and then pushes the command object onto the history stack. An undo operation takes the top item from the history stack and invokes undo() to reverse the operation. The reversed object can then be pushed on the redo stack so that if the user requests a redo operation the controller takes the top command from the redo stack, invokes doIt() and pushes the command back onto the undo stack.

```
public interface Command
{
	public void doIt();	// execute the command
	public void undo();	// undo the command
}
```

Figure 16.4 – Command object interface

In this approach each command object class is responsible for remembering the required information for doIt() and undo() to be performed. The general undo mechanism only needs to know about the Command interface.

The model shown in figure 16.5 is for a simple text editor. There are operations to insert and delete pieces of a string based on the index of the location where the edits are to occur.

```
public class TextModel
{
        public int nCharacters() { .. }    // returns the number of characters in the string
        public String getCharacters(int firstIdx, int lastIdx) { .. }
                // returns the characters stored at firstIdx through lastIdx inclusive.
        public void delete(int firstIdx, int lastIdx) { .. }
                // deletes the characters at firstIdx through lastIdx inclusive.
        public void insert(int firstIdx, String newChars) { .. }
                // inserts newChars into the text so that the first character at newChars will
                // be at firstIdx.
}
```

Figure 16.5 – Text editor model

Figure 16.6 shows the command objects for delete and for insert. Each has three parts: 1) a constructor that acquires the model and the parameters for the command, 2) a doIt() method to perform the command and save any necessary information, and 3) an undo() method to restore the model. It is possible to create these command object classes in such a way that the original methods can be made private and accessible only to the objects. This ensures that controller code only uses the undoable approach for making model changes.

```
public class DeleteCmd implements Command
{       private int firstIdx, lastIdx;
        private String deletedChars;
        private TextModel model;

        public DeleteCmd(TextModel model, int firstIdx, int lastIdx)
        {       deletedChars=null;
                this.model=model;
                this.firstIdx=firstIdx;
                this.lastIdx=lastIdx;
        }
        public void doIt()
        {       deletedChars=model.getCharacters(firstIdx, lastIdx);
                model.delete(firstIdx, lastIdx);
        }
        public void undo()
        {       model.insert(firstIdx,deletedChars);  }
}
public class InsertCmd implements Command
{       private int insertIdx;
        private String insertChars;
        private TextModel model;

        public InsertCmd(TextModel model, int insertIdx, String insertChars)
        {       this.insertIdx=insertIdx;
                this.insertChars=insertChars;
                this.model = model;
        }
        public void doIt()
        {       model.insert(insertIdx,insertChars); }
        public void undo()
        {       model.delete(insertIdx, insertIdx+insertChars.length()-1); }
}
```

Figure 16.6 – Command objects for TextModel

Suppose that our controller has a variable called cursor that stores the current insertion point for our text editor application. Suppose also that there is an undoHistory object that stores previously executed commands. Based on this we can write the controller's keyPressed() method for accepting keyboard input. The method will insert characters, delete characters on backspace and perform undo when a control-Z is received. The code for this method is shown in figure 16.7 and demonstrates how the controller would use command objects to perform actions and to get them undone. We could simplify this code by having the undoHistory.push() method perform the doIt() before placing the command on the stack.

```
public void keyPressed(Char keyCharacter)
{
        Command cmnd=null;
        if ( isBackspace(keyCharacter) )
        {       cmnd=new DeleteCmd(myModel,cursor,cursor);
                cmnd.doIt();
                undoHistory.push(cmnd);
        }
        else if ( isControlZ(keyCharacter) )
        {       cmnd=undoHistory.pop();
                cmnd.undo();
        }
        else
        {       cmnd=new InsertCmd(myModel,cursor,new String(keyCharacter));
                cmnd.doIt();
                undoHistory.push(cmnd);
        }
}
```

Figure 16.7 – Implementing keyPress() using command objects

As was mentioned earlier some controller actions will cause multiple model changes that we may want to undo as a single unit. We can handle this by creating a special CommandList class that extends Command. This would have an add(Command) method that would add command objects to the list. The CommandList.doIt() would perform doIt() on each command object in its list in order. The CommandList.undo() would perform undo() on each command object in reverse order. Using CommandList a controller can collect any number of model changes and then push them on the history stack as one command to be undone.

Scripting with Command objects

A scripting system can be built from command objects in much the same way as the action log. We need to add the method cmndString() to our Command interface. This method will convert the command object into a string that can be written to a script file. We also need to parse the command. When given a command string we first need to identify which command object class should be instantiated. Remember that the different command behaviors are associated with different classes of objects. A simple technique is to create a hash table that maps command names to object classes. We then would have command strings of the form:

commandName parameters

To parse a command string we first extract the command name, look up the appropriate class in the table and instantiate the correct class of Command object. We need to add a parseParameters() method to Command so that we can pass the remainder of the command line to the object. Parsing the parameters should fill the command object with the information that it needs. We then execute the command by calling doIt(). It is common to have a generic parser that separates the parameters into an array of strings. The parseParameters() method only needs to interpret each parameters string. This command formulation shares the TCL structure in many ways.

To make parseParameters() work we need to pass in a reference to all of the models in the application. We will also need a mechanism for referencing parts of the model for use as parameters to the commands. Generally this is done by some kind of path expression such as document.paragraph[7].10-12 to select characters 10 through 12 of the seventh paragraph in the document. The nature of these path expressions will vary from application to application. This is a key technique from the Domain Object Model (DOM) used in JavaScript[6]. If every model supports a getPathObject(String path) and a getPathString(Object modelObject) method then we have what we need. A reference to any object can be obtained by getPathString() and added to the parameter string in cmndString() and when parsed out of the string by parseParameters() the reference can be used as an argument to getPathObject() to retrieve the desired object. Object paths plus strings and numbers will generally provide everything necessary to formulate and parse command strings using the Command object architecture. Figure 16.8 shows an example of how this might be done for our simple insert/delete model.

```
public class DeleteCmd implements Command
{       private int firstIdx, lastIdx;
        private String deletedChars;
        private TextModel model;

        public DeleteCmd(TextModel model, int firstIdx, int lastIdx)
        {       code to build object from parameters }
        public DeleteCmd() { everything to null }
        public void doIt() { . . . }
        public void undo() { . . . }
        public String cmndString()
        {       return “delete “+model.getPathString()+” “+firstIdx+” “+lastIdx; }
        public void parseParameters(String args[])
        {       model = (TextModel) getPathObject(args[0]);
                firstIdx= Integer.parse(args[1]);
                lastIdx= Integer.parse(args[2]);
        }
}
public class InsertCmd implements Command
{       private int insertIdx;
        private String insertChars;
        private TextModel model;

        public InsertCmd(TextModel model, int insertIdx, String insertChars)
        {       code to build from parameters }
        public InsertCmd() { set everything to null }
        public void doIt() { . . . }
        public void undo() { . . . }
        public String cmndString()
        {       return “insert “+model.getPathString()+” “+insertIdx+” ‘””+insertChars+”\””; }
        public void parseParameters(String args[])
        {       model= (TextModel) getPathObject(args[0]);
                insertIdx=Integer.parse(args[1]);
                insertChars=args[2]; // quotes already removed by parameter parser
        }
}
```

Figure 16.8 – Scripting with Command objects

This is not the only way scripting languages can be organized. A command string could start with a path expression and then the object retrieved could have a makeCommand(String commandName) method to produce the desired Command object and attach it to the model. The remaining parameters are then handled as before. This provides a more object-oriented scripting language.

Representing the history

In many applications, the history stack is invisible to the user. All that the user sees is the undo command in the menu or a special key such as control-Z.

With an invisible history it is difficult to make effective use of a large history stack. The user must remember a desired point in history and accurately return to it. The redo command is helpful here. If, as commands are undone, they are placed on the redo stack then if the user goes back too far then can invoke redo to undo the undo. In many applications is it helpful to show the history stack to the user. Figure 16.9 shows the history stack from Adobe Photoshop.

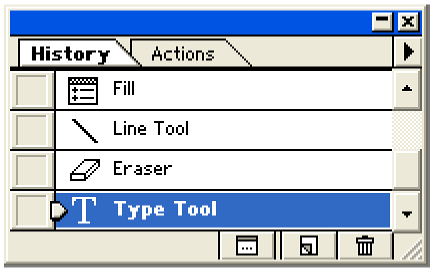


Figure 16.9 – PhotoShop history

This kind of history presentation is easily created by adding the getIcon() and getName() methods to Command. These methods provide for the display of the history stack shown in figure 16.9. In addition to the standard undo command the user can select any command in the history stack display. This will cause all commands that are later than the selected command to be undone in order. In PhotoShop the undone commands are displayed in gray to show that they are undone. Selecting an undone command causes redo operations up to the point of the selected command. These operations are quite simple to implement given the Command object architecture. The display of the history stack provides a rather friendly way to undo arbitrary amounts of work.

Versioning

It is frequently helpful for applications to maintain versions of a model so that changes can be tracked. This is quite common for source code control systems such as CVS[7]. When many people are working on a project or the work is extensive it is helpful to break it up into *patches* or *versions*. This allows the project to roll back to a previous state where everything was stable. Most versioning systems are based on the forward undo model. A baseline version of the object model is stored and then *patches* are stored that will convert the baseline from version to version.

Patches can be constructed in two ways: differencing and history scripting. The differencing approach takes two versions of the model: before and after. A difference method is applied to this pair to produce a description of how to change the before model into the after model. For source code control this difference is a list of insert/delete commands. The two versions of the source code are compared and a list of insert/delete edits are generated. If we are using a Command object architecture this approach can be applied to any model, not just text. Two versions of the model are compared and a list of Command objects generated that will convert before into after. Using our cmndString() method we can generate a text file that contains the patch that will convert before into after. We can apply the patch by loading the baseline model and executing the command objects in each patch until the desired version is obtained. All that is needed to implement this difference form of versioning is a difference() method that can be applied to two versions of the same model. This difference() method is sometimes difficult to implement.

The history scripting approach to versioning saves the history stack as a command script. Producing a desired version involves loading the baseline model and applying the saved history commands to produce the new model. This is much simpler to implement because we just use the existing mechanisms with no need to implement a difference() method. The only drawback is that this may produce less space efficient scripts. For example constructing a paragraph by replaying insert commands for every character is not as efficient as recognizing a large insertion and generating one command that does it all. This can be mitigated by some of the history granularity issues described earlier in the chapter. The history approach also remembers all of the intermediate false steps that a user may have gone through to reach the final version. The differencing technique takes a shorter path between versions without replaying the user's uncertainty and explorations. Either differencing or history scripting with command objects can provide versioning capabilities to any application.

For such a versioning mechanism to be really effective there must be a user interface that clearly shows the differences between two versions by showing the changes made by each of the commands in the patch. This is a more difficult problem than tracking the versions themselves. Such solutions are readily available for text such as programs but not as common for other interactive information.

Summary

If we track every command that a user may generate, we can the undo any of those commands. This creates a safe and more user-friendly environment. The Command object architecture is the most robust for this purpose. Once we have begun saving objects for every interactive command we have enough information to generate a scripting language for that application and a versioning mechanism to support group work.

---

[1] Shneiderman, Ben, Direct manipulation: A step beyond programming languages, *IEEE Computer 16*, 8 (1983), 57–69.

[2] Berliner, B. "CVS II: Parallelizing Software Development," *Proceedings of the USENIX Winter 1990 Technical Conference*, (1990), pp 341-352.

[3] Barron, D. *The World of Scripting Languages*, John Wiley & Sons, (2000).

[4] Ousterhout, J. *TCL and the TK Toolkit*, Addison-Wesley, (1994).

[5] Meyer, B. *Object-Oriented Software Construction*, Prentice-Hall (1988).

[6] Flanagan, D. *JavaScript: The Definitive Guide*, O'Reilly Media Inc (2001).

[7] www.CVSHome.org