Implementing a Selective Undo Framework in Python

Monty Zukowski
Zero to One Software Design
jamz@cdsnet.net

Abstract

I've implemented in Python the undo algorithm found in "Undoing Actions in Collaborative Work: Framework and Experience" by Prakash and Knister [PK94]. Their approach helps structure the objects that make up a "document". Those same objects are well suited for use in an embedded language like Python: with them you can write scripts that can be undone! I've implemented standard file operations as a simple concrete example of using the framework.

1 Introduction

I've always wondered how to design the objects which work together to represent a document, the *model* of the model-view-controller paradigm. How to design them to be easily scriptable? Undoable? Well, when I stumbled onto "Undoing Actions in Collaborative Work: Framework and Experience" by Prakash and Knister [PK94] I struck a goldmine. Their paper gives a great overview of undo algorithms and when and why to use them. The "selective undo" algorithm presented by them lets the user undo *any* previous action, not just the last. So in a groupware application, she or he can undo her or his last action, not just the last action applied to the document by someone else.

As an adminstrator of hundreds of thousands of multimedia files, I wrote many Python scripts to move them around, rename them and tweak them for our build process to turn them into CD-ROMs. Every now and then I'd get cocky, run a script without testing it, and rename all the files to the same filename, or do other stupid file tricks. "If only I could undo that script!" I would think. Well, now I can since I've written undoable copy, move, link and delete operations as a simple application of the selective undo algorithm.

At first I sat down to write a general purpose undo module. You, the programmer, could just drop it into your application, add some glue, and get undo for free. I wish! It turned out that how the undo algorithm works is tightly linked to how the building block objects are designed. Or rather the other way around: the building block objects need to be designed to fit into the undo framework. Since undo is one of those things I would normally want in an application, I was glad to learn how to write objects that behave well with the undo algorithm.

I have written a general purpose `History` module which implements the selective undo algorithm. It isn't and cannot be a black box module. You need to understand how it works in order too use it. That's why I'm presenting it as a *literate program* as described by Knuth [Knu92] to explain how it works. Literate programming lets me write my code and documentation in one file and in any order I want, so I can explain what I'm doing in the order I think best, not the order imposed by the programming language. The same literate program file creates this document as well as the machine readable source code.

1.1 Selective Undo Overview

Please do read Prakash and Knister [PK94] to understand *why* their algorithm works. Here's a quick overview of *how* it works. Define operations that affect your state (the document). All such operations must define an inverse operation. Applying this inverse immediately after the operation nullifies it and returns the document to the state before applying the operation. At this point you have the Command pattern as described in *Design Patterns* [EG94, pages 233-242].

If you, the programmer, want to undo an operation that was not the last one, you can't always get away with applying the inverse of that operation to the current state. Say my document is the sentence "Python rocks!". I select "rocks" and type in "rules". Call that *op1*, an `InsertText` operation which stores the selection which was modified: in this case chars `[7:12]`, the inserted text "rules" and the replaced text "rocks". The `InsertText` object needs the replaced text so it can create the proper inverse operation, which in this case would be `InsertText([7:12], "rocks")`. Next I type "really " in front of "rules". This will be *op2*, another `InsertText` operation with selection of `[7:7]`, inserted text of "really ", and replaced text "". Now what's required to undo *op1*?

Just applying the inverse of *op1* to the current document of "Python really rules!" would result in "Python rocksy rules!" because "rules" moved without *op1* knowing about it. The approach in Prakash and Knister [PK94] is to have a Transpose(a,b) function which modifies a to make it as if a was applied after b, instead of before b. In the case of *op1* and *op2*, transposing would involve recognizing that op2 shifted any character positions >= 7 by 7 characters. That would mean the range [7:12] of *op1* would change to [14:19]. Now the inverse of *op1* would correctly create "Python really rocks!".

In practice the algorithm doesn't actually apply the Transpose() function to the history list directly. Instead it copies the list and transposes the operation to undo to the end. Since the history list includes everything done to a document, the end of the history list represents the current state of the document. Then the algorithm takes the inverse of the transposed operation and applies that to the document.

There are cases where a previous operation can't be undone. What if after *op2*, instead of undoing *op1*, I delete the entire sentence as *op3*? After that it wouldn't make much sense to revert "rules" to "rocks" since "rules" no longer exists! To detect such conflicts, a Conflict(a,b) function is defined. The reason the algorithm needs the Conflict(a,b) function is because not all conflicts will prevent an undo. Consider applying *op4* as the inverse of *op3*. There is still a conflict between *op1* and *op3*, but *op4* cancels out *op3*, so it still should be able to undo *op1*. A method called removeDoUndoPair handles these situations.

In summary the algorithm to undo any previous operation first copies the history list from the point of the operation to be undone. Then it transposes that operation to the end of the list copy. This brings the operation into the current state of the document. Then it takes the inverse of that transposed operation and applies it to the document to undo the original operation.

2 The History Module

The History module defines three objects: History, the actual history list, HistoryNode, a single element in the history list, and AbstractOperation, which you, the programmer, subclass to create your own document operations.

Literate programs define code in terms of "chunks". The notation below says that the "History.py" chunk is defined as the four following chunks. Chunks can have pieces of code in them as well. Chunks can also be appended to, as with the HistoryExceptions chunk, which I add on to whenever I need to define a new exception.

```
⟨History.py⟩≡
  ⟨HistoryExceptions⟩
  ⟨HistoryNode⟩
  ⟨AbstractOperation⟩
  ⟨History⟩
```

2.1 Abstract Operation

Now the fun part, AbstractOperation. It's the base for operations which work on a document. These operations must define Conflict, Transpose, inverse, and perform methods. AbstractOperation defines Conflict(A,B) and Transpose(A,B) in a very simplistic way. Any serious subclass will certainly override these methods to be more efficient. Conflict checks to see what attributes are affected by each operation and signals a conflict if they overlap. Transpose just tests for a conflict and if there is none returns the identical objects swapped. The inverse() and perform() methods are also defined and must be implemented by subclasses to provide the inverse operation and to actually modify the document.

Initially I had hoped to be able to mix various primitive operations together and have them register what they affect in a table which could be used to deduce the Conflict and Transpose functions. But to implement efficient Conflict and Transpose routines they really have to have detailed knowledge about how all operations are implemented. These functions violate the encapsulation of operations because they need to know how the operations represent themselves and what they do to a document. Conceptually, operations are methods of some Document object, even though I implement them here as independent objects. To implement Conflict and Transpose you need to know everything about all operations. You can't just add in a new operation without thinking through how it will affect all the other operations. This may not be as limiting as you might think. You can easily implement higher-level actions out of the existing operations and be able to undo them since they only affected the document through undoable operations!

```
⟨HistoryExceptions⟩≡
  UnimplementedBySubclass = "This method must be\
  implemented by a subclass"
```

```
⟨AbstractOperation⟩≡
  class AbstractOperation:

    def perform(self,context=None):
      return

    def readSet(self):
      return ()

    def writeSet(self):
      return ()

    def inverse(self):
      raise UnimplementedBySubclass

    def copy(self):
      raise UnimplementedBySubclass

    def Transpose(a, b):
      return (b, a)
```

```
def Conflict(a, b):
  if a is None or b is None:
    return None
  bw = b.writeSet()
  br = b.readSet()

  # b conflicts with a if it reads or writes
  # anything a writes
  for w in a.writeSet():
    if w in bw or w in br:
      return 1

  #b conflicts with a if it writes anything a reads
  for w in a.readSet():
    if w in bw:
      return 1
  return None

def isNullOp(self):
  return 1
```

2.2 History

Once you have the details of your primitive operations worked out, you can build composite operations from them and know that they will be undoable. I had originally structured the History class as a tree so that composite operations would have the primitive operations as children nodes. That way it would be easy to recognize and undo an entire composite operation at once by undoing all of its children. Having the History as a tree, however, turned the implementation into a mess. I had defined beginGroup() and endGroup() methods and left it to the programmer to use them properly. Later I realized that it would be impossible to have a coherent tree structure if in fact multiple people edited a document simultaneously and grouping operations were not atomic between the beginGroup() and endGroup(). I decided I didn't need that headache and turned History into a list with callbacks that could let programmers keep their own tree and also keep any other tables of HistoryNodes they want. For instance I could keep a table of file transfers that were for a particular product, or maybe define an experimental mode when the user is playing around and may want to return to a previous state by rejecting the entire experiment.

The History object keeps a list of the operations applied to a document, provides the actual undo routine, a method to call whenever an operation has been applied to a document, and callbacks for notification of do and undo actions. The History attributes follow.

context	the object that applies an operation to a document
historyList	the list of HistoryNodes
historyNodeClass	the class used for elements of the list
callbacks	functions to call whenever a node is added

Callback functions must take a History object and HistoryNode object as their first two arguments.

⟨*History*⟩≡

```
class History:
  def __init__( self,
               context,
               historyList = None,
               historyNodeClass = HistoryNode,
               callbacks = None ):
    self.context = context
    if historyList is not None:
      self.historyList = historyList
    else:
      self.historyList = []
    self.historyNodeClass = historyNodeClass
    if callbacks is not None:
      self.callbacks = callbacks
    else:
      self.callbacks = []

  def addHistoryOp(self, op):
    historyNode = self.historyNodeClass(op)
    self.historyList.append(historyNode)
    for call in self.callbacks:
      call(self, historyNode)

  def addCallback(self, callback):
    self.callbacks.append(callback)

  def removeCallback(self, callback):
    self.callbacks.remove(callback)

  ⟨undo routines⟩
```

2.3 HistoryNode

The HistoryNode holds some special information for the undo algorithm, the operation that was performed, and any other attributes you want to add in via a callback, such as username or timestamp. It can also make a copy of the undo related attributes for use in the undo routines. The undo related attributes follow.

op	the operation performed
undoBlockMember	whether this is part of a nested do/undo block
undoneBy	the HistoryNode which undid this operation
undid	the HistoryNode which this operation undid

⟨*HistoryNode*⟩≡

```
class HistoryNode:

  def __init__ (  self,
                op,
                undoBlockMember = None,
                undoneBy = None,
                undid = None) :

    self.op = op
    self.undoBlockMember = undoBlockMember
    self.undoneBy = undoneBy
    self.undid = undid
```

```
  def tempCopy(self):
    return self.__class__(  self.op.copy(),
             undoBlockMember = self.undoBlockMember,
             undoneBy = self.undoneBy,
             undid = self.undid)
```

2.4 Undo

The undo algorithm is actually quite simple. It makes a copy of all operations starting after the one to undo and ending with the last operation done on the document. Then it moves the undo operation to the end of the list by transposing elements, checking for conflicts along the way. Once it is at the end of the list, it takes the inverse and applies it to the document. Actually there is another detail: if an operation in the list copy is already undone and conflicts with the operation to undo, there really isn't a conflict since the offending operation was undone already. In that case it can remove the do-undo pair from the copy of the list.

⟨*HistoryExceptions*⟩+≡

```
  UndoConflict = "Undo conflict, incompatible operations"
```

⟨*undo routines*⟩≡

  ⟨*undo*⟩
  ⟨*removeDoUndoPair*⟩
  ⟨*tempListForUndo*⟩

2.4.1 tempListForUndo

The tempListForUndo method constructs a copy of self.historyList starting just after the node to be undone, copying each HistoryNode by calling its tempCopy() method. I keep a flag in HistoryNode for nested blocks of do-undo's. ABCDD'C'B'A' is effectively a null operation, where the primes signify undo operations. As such it can be ignored completely, so I skip over such blocks when making the temporary list. I also return a dictionary with new HistoryNode copies as keys and old HistoryNodes as values so any exceptions can refer to the original nodes to get all the attributes stored in it which may not be in the copy.

⟨*HistoryExceptions*⟩+≡

```
  NoHistNode = "historyNode not found in historyList"
```

⟨*tempListForUndo*⟩≡

```
  def tempListForUndo(self, historyNode):
    if historyNode not in self.historyList:
      raise NoHistNode, (historyNode, self)

    histList = self.historyList[
             self.historyList.index(historyNode)+1:]
    copy = []

    newToOldMap = {}
    oldToNewMap = {}

    skipUntil = None
    for node in histList:
      if (not skipUntil and node.undoBlockMember
          and node.undoneBy):
        skipUntil = node.undoneBy
      if skipUntil and node != skipUntil:
        continue
      elif node is skipUntil:
        skipUntil = None
        continue  # we still want to skip this node
      newnode = node.tempCopy()
      copy.append(newnode)
      newToOldMap[newnode] = node
      oldToNewMap[node] = newnode
    for node in copy:
      if node.undoneBy:
        node.undoneBy = oldToNewMap[node.undoneBy]
      if node.undid and oldToNewMap.has_key(node.undid):
        node.undid = oldToNewMap[node.undid]
    return copy, newToOldMap
```

2.4.2 removeDoUndoPair

The removeDoUndoPair method modifies nodes in the list passed in to cancel out a *do-undo* pair by transposing the *do* until it is next to the *undo*. Whey they are together they nullify, which it signals by setting the op attribute to None for both of them. It does know for sure though that there are no real conflicts between the *do* and *undo*, otherwise the *do* could not have been undone in the first place! So if any conflicts are found it knows that the offending operation has already been undone and calls removeDoUndoPair() recursively to eliminate it.

⟨*removeDoUndoPair*⟩≡

```
  def removeDoUndoPair(self, tempList):
    doNode = tempList[0]
    for index in range(1, len(tempList)):
      node = tempList[index]
      if node is doNode.undoneBy:
        break
      if node.op is None:
        continue
      if doNode.op.Conflict(node.op):
        self.removeDoUndoPair(tempList[index:])
      else:
        node.op, doNode.op = doNode.op.Transpose(node.op)
    doNode.op = node.op = None
```

2.4.3 undo

With all the support for the undo algorithm in place, it practially writes itself! It gets a copy of the history list to work with, shifts the operation to undo to the end of the list, then performs its inverse and keeps track of who undid who.

⟨*undo*⟩≡

```
  def undo(self, historyNode):

    tempList, newToOldMap = self.tempListForUndo(
                      historyNode)
    shiftOp = historyNode.op.copy()

    index = 0
```

```
#move shiftOp to end of history list
for node in tempList:
  if node.op is None or node.op.isNullOp():
    continue
  if shiftOp.Conflict(node.op):
    if node.undoneBy:
      self.removeDoUndoPair(tempList[index:])
    else:
      raise UndoConflict,(historyNode,newToOldMap[node])
  else:
    unused, shiftOp = shiftOp.Transpose(node.op)
  index = index + 1

#perform the inverse,
#context will call us back to add it to History
self.context.performOp(shiftOp.inverse())
endNode = self.historyList[-1]
historyNode.undoneBy = endNode
endNode.undid = historyNode
if not tempList:  #means all ops between historyNode and
                  #end cancelled out
  endNode.undoBlockMember = 1
  historyNode.undoBlockMember = 1
```

Conclusion: Guidelines for building document objects and possible uses

So what have I learned about structuring document objects for undo? One thing is that it is not easy to structure the undo objects into a separate module. They need to be tied directly to the implementation of the operations that modify documents because the document data and the operations that work on it need to know about each other to be able to define Conflict and Transpose properly.

So now the guidelines. First, come up with your state or document representation and what operations you want to change that state with. Keep in mind that to be undoable you must provide an inverse for each operation. Usually this means squirrelling away whatever that operation destroys so the inverse can recreate it. You will also need to define Conflict and Transpose functions that work over all operations. With these features designed in at the start you'll have everything the selective undo algorithm needs to work. Prakash and Knister [PK94] gives some excellent examples of implementations of this framework.

The prototypical modules I've presented do work. I've tested them but have not used them extensively, so they are not mature by any means. In a real application I would improve them by customizing or subclassing AbstractOperation. The readSet and writeSet approach may not be the correct way to determine conflicts for the document representation. At the very least I would change them to use real Set objects. The context and callback mechanisms for actually performing the operations and notification of additions to the history list may need improvements for performance reasons. And, with the History module working to satisfaction now, it could be moved into C or Java for performance reasons.

Note that once you have all your operations behaving as described above you've nearly got a domain specific language! You or your users can write functions using these operations and get undo for free. With a little extra bookeeping you could record which function invocation caused which operations in the history list and use that information to undo the entire thing at once. Your operations and document object represent a clean interface to your documents and should allow for clear separation between your UI and document object. In addition to making it easy to embed a language like Python in your application for macros, it should also be easy for programs, even remote programs, to manipulate documents from outside the application.

For groupware applications, the objects would be written the same way, but the context object would have to synchronize with a central server. The Conflict function would help identify problems when users simultaneously do things that aren't compatible. Prakash and Knister [PK94] goes into detail about these issues.

"Extending programming by demosnstration with hierarchical event histories" by Kosbie and Myers [KM94] discusses techniques for analyzing user's histories in order to facilitate "programming by demonstration". They recommend keeping histories as a tree to help the analysis. This could be done fairly easily using a callback and keeping the tree structure external to the History object. You could then use their techniques to try to "learn" what the user is doing. Or you could just have that information browsable to help the user write macros based on what they have been doing.

Ever since I started programming graphical user interfaces I've been curious about how to implement undo and macro languages. Those topics have been the holy grail which I've quested after. When I learned about Python a few years ago, I found the embeddable language I was looking for. With Prakash and Knister's paper [PK94] I've finally discovered and implemented a serious undo framework. Their approach is like defining your own algebra for manipulating documents because it lets you switch around events in the history so you can undo anything, not just the last action. Now I'm ready to do a really killer app! Any ideas?

Availability

The source code for History.py and UndoableFileOperations.py will be available by the time this paper is published. Check either on the Python home page, http://www.python.org or look for my quarters on the

Python starship, http://starship.skyport.net/crew.html.

References

[EG94] Erich Gamma et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 94.

[KM94] David S. Kosbie and Brad A. Myers. Extending programming by demonstration with hierarchical event histories. Technical Report CMU-HCII-94-102, Carnegie Mellon University, The Human Computer Interaction Institute, May 94.

[Knu92] D.E. Knuth. *Literate Programming*. Stanford University, 92.

[PK94] Atul Prakash and Michael J. Knister. Undoing actions in collaborative work: Framework and experience. Technical Report CSE-TR-196-94, University of Michigan, March 94.

Appendix: Undoable File Operations

As a final example I've implemented some simple undoable file operations. They all move files that would be deleted into a trash folder and remember that they did so, so they can be undone later if need be. They demonstrate how to use the readSet and writeSet routines for conflicts. A simple test routine creates some files, deletes them all, then undoes the first delete.

⟨*UndoableFileOps.py*⟩≡

```
import os, History

class UndoableBinaryFileOp(History.AbstractOperation):
 def __init__(self, frm, to, trashDir = None, trashed = None):
  self.frm = frm
  self.to = to
  self.op = None
  self.trashDir = trashDir
  self.trashed = trashed
 def perform(self):
  if os.path.isfile(self.to) and self.trashDir is not None:
   self.trashed = getUniqueName(self.to, self.trashDir)
   os.rename(self.to, self.trashed)
  else:
   self.trashed = None
  apply(self.op, (self.frm, self.to))

 def readSet(self):
  return (self.frm, )
 def writeSet(self):
  if self.trashed:
   ret = (self.frm, self.to, self.trashed)
  else:
   ret = (self.frm, self.to)
  return ret
 def inverse(self):
  return None
 def isNullOp(self):
  return 0

class UndoableMove(UndoableBinaryFileOp):
 def __init__(self, frm, to, trashDir = None, trashed = None):
  UndoableBinaryFileOp.__init__(self, frm, to,
                trashDir, trashed)
  self.op = os.rename
 def inverse(self):
  if self.trashed:
   ret = AtomicOpsTuple((UndoableMove(self.to, self.frm),
      UndoableMove(self.trashed, self.to, self.trashDir)))
  else:
   ret = UndoableMove(self.to, self.frm)
  return ret
 def readSet(self):
  return (self.frm, self.to)
 def copy(self):
  return UndoableMove(self.frm, self.to,
           self.trashDir, self.trashed)

class UndoableCopy(UndoableBinaryFileOp):
 def __init__(self, frm, to, trashDir = None, trashed = None):
  UndoableBinaryFileOp.__init__(self, frm, to,
                trashDir, trashed)
  self.op = os.copy
 def inverse(self):
  if (self.trashed):
   return UndoableMove(self.trashed,
            self.to, self.trashDir)
  else:
   return UndoableDelete(self.to,self.trashDir)
 def copy(self):
  return UndoableCopy(self.frm, self.to,
           self.trashDir, self.trashed)

class UndoableSymlink(UndoableBinaryFileOp):
 def __init__(self, frm, to, trashDir = None, trashed = None):
  UndoableBinaryFileOp.__init__(self, frm, to,
                trashDir, trashed)
  self.op = os.symlink

 def inverse(self):
  if (self.trashed):
   return UndoableMove(self.trashed,
            self.to, self.trashDir)
  else:
   return UndoableDelete(self.to,self.trashDir)

 def copy(self):
  return UndoableSymlink(self.frm, self.to,
            self.trashDir, self.trashed)

class UndoableHardlink(UndoableBinaryFileOp):
 def __init__(self, frm, to, trashDir = None, trashed = None):
  UndoableBinaryFileOp.__init__(self, frm, to,
                trashDir, trashed)
  self.op = os.link

 def inverse(self):
  if (self.trashed):
   return UndoableMove(self.trashed,
            self.to, self.trashDir)
  else:
   return UndoableDelete(self.to,self.trashDir)
```

```
  def copy(self):
    return UndoableHardlink(self.frm, self.to,
                            self.trashDir, self.trashed)

class UndoableDelete(History.AbstractOperation):
  def __init__(self, filename, trashDir = None, trashed = None):
    self.filename = filename
    self.trashDir = trashDir
    self.trashed = trashed

  def copy(self):
    return UndoableDelete(self.filename,
                          self.trashDir, self.trashed)

  def perform(self):
    if (os.path.isfile(self.filename)
       and self.trashDir is not None):
      self.trashed = getUniqueName(self.filename,
                                   self.trashDir)
    else:
      self.trashed = None
    if self.trashed:
      os.rename(self.filename, self.trashed)
    else:
      os.unlink(self.filename)

  def readSet(self):
    return (self.filename,)

  def writeSet(self):
    if self.trashed:
      ret = (self.filename, self.trashed)
    else:
      ret = (self.filename)
    return ret

  def inverse(self):
    return UndoableMove(self.trashed,
                        self.filename, self.trashDir)

  def isNullOp(self):
    return 0

class AtomicOpsTuple(History.AbstractOperation):
  def __init__(self, operationsTuple):
    self.operationsTuple = operationsTuple

  def copy(self):
    return AtomicOpsTuple(self.operationsTuple)

  def perform(self):
    for op in self.operationsTuple:
      op.perform()

  def readSet(self):
    ret = ()
    for op in self.operationsTuple:
      ret = ret + op.readSet()
    return ret

  def writeSet(self):
    ret = ()
    for op in self.operationsTuple:
      ret = ret + op.writeSet()
    return ret

  def isNullOp(self):
    return 0

def getUniqueName(fn, trashDir):
  base = os.path.basename(fn)
  count = 1
  result = os.path.join(trashDir, base + '%.3d' % count)
  while os.path.isfile(result):
    count = count + 1
    result = os.path.join(trashDir, base + '%.3d' % count)
  return result

if __name__=='__main__':
  limit = 12

  #a helper function to see what's going on
  def listem():
    print
    for dir in ["/tmp/tests", "/tmp/tests/trash"]:
      l = os.listdir(dir)
      for f in l:
        fn = dir+'/' + f
        if not os.path.isfile(fn):
          print '\t' + fn
        else:
          fi = open(fn)
          s = fi.read()
          fi.close()
          print '\t' + fn, s

  #create Context and History objects
  class Context:
    def performOp(self, op):
      op.perform()
      self.history.addHistoryOp(op)
  c = Context()
  h = History.History(c)
  c.history = h

  #create some directories and files to play with
  try:
    os.mkdir("/tmp/tests",0777)
  except:
    pass

  try:
    os.mkdir("/tmp/tests/trash", 0777)
  except:
    pass

  for x in range(limit):
    f = open('/tmp/tests/hey'+str(x), 'w')
    f.write(str(x))
    f.close()

  listem()

  #delete everything
  for x in range(limit):
    u = UndoableDelete('/tmp/tests/hey'+str(x),
                       '/tmp/tests/trash')
    c.performOp(u)

  listem()

  #undo only the first delete
  h.undo(h.historyList[0])

  listem()
```