NXApp, Summer 1994. Volume 1, Issue 3.

A Methodology for Message-Based Undo and Animation

written by **Jeff Martin**

The ability to undo actions is an important and useful feature in any application, but it can prove difficult to implement. The UndoManager object provides a mechanism for this feature by taking advantage of the message dispatcher in Objective C. This technique can also be used to implement simple animation.

NEEDED BUT NEGLECTED

Of all the features that really polish a NEXTSTEP application, like services, hot links, and preferences, one is often left undoneÐundo.

There's a good reason for this: Undo is the feature that never ends. For every new action that you add to your application, you have to follow it up with code to undo that action. This can sometimes double the work required to add a new feature.

My approach to undoÐmotivated originally by a desire for simple animation, as you'll see

later in the articleÐis based on the premise that undoable actions are usually invoked at a method
level and that the most appropriate place to undo the effects of a method is in the method itself. Objective C provides a unique environment for trapping and storing method invocations to perform later. I've implemented an object that takes advantage of this feature in the UndoManager MiniExample.

UndoManager is available as NeXTanswers #1582.

THE BASICS OF UNDOMANAGER

My goal was to be able to take an undoable action method and implement undo with one message call from within that method. For instance, consider a method that sets the color of an object:

```
- setColor:(NXColor)value
{
   // Set the new color value and return
   color = value;
   return self;
}
```

This method not only sets the new color value, but also is aware of the old color value. Therefore, this method could easily send a message to a central object (the UndoManager object) that would registerº this change. Here's an example:

```
- setColor:(NXColor)value
{
   // Register '-setColor:' method and the old color value
   // with the UndoManager
```

```
    [undoManager setColor:color];

    // Set the new color value and return
    color = value;
    return self;
}
```

When the UndoManager object receives this message it can store it away and later invoke it, for example when the user chooses an Undo command from a menu.

This suggests that an UndoManager object would have to implement the method **setColor:** to catch this change. However, we can instead use a feature of the Objective C message dispatcher, the **forward::** method. The **forward::** method is a convenient method that's invoked whenever you send a message to an object that doesn't understand it. You may have invoked its default implementation if you have ever seen the message ªObject so-and-so does not respond to method such-and-such.º (Of course, I've never had this happen to meÐI've just heard about it.)

forward:: takes two arguments. The first is the selector or method that was called, and the second is a nice little package of its arguments that can easily be copied, stored, and later invoked with the **performv::** method.

To find out more about the **forward::** method, see the specifications for the Object class.

Unfortunately the Objective C run-time system doesn't provide a legal way to discern the sender of a message. This means that for an UndoManager object to later call a message that was registered, it needs for the target of that message to be registered along with the message. This can be done with a **setUndoTarget:** method. With this addition, our example **setColor:** method now becomes this:

```
- setColor:(NXColor)value
{
   // Register self, the '-setColor:' method
   // and the old color value with the UndoManager
   [[undoManager setUndoTarget:self] setColor:color];

   // Set the new color value and return
   color = value;
   return self;
}
```

Therefore the core of our UndoManager object manages a list of (*target*, *action*, *argument*) records in an undoList and implements the following methods:

- **setUndoTarget:** Sets which object should be associated with any messages that are registered following its invocation
- **forward::** Captures and stores the actions and their arguments
- **undo:** Pops the last undoable action from the list and performs it

MULTIPLE LEVELS OF UNDO

Users prefer multiple levels of undo, so they can undo a whole series of steps, not just the most recent one. From an implementation standpoint there needs to be a limit to the levels of undo that are stored, or the app will quickly take up a lot of space when it runs. Since the UndoManager stores a list of undoable records, it would be pretty straightforward to check the length of the list when a new record is added. If the length exceeds a set amount, the record at the beginning of the list can be removed and freed. The length can be set and queried with methods like **(int)levelsOfUndo** and **setLevelsOfUndo:(int)***value*.

This may seem like a simple, straightforward feature, but once the UndoManager starts freeing references to data structures, there are memory management issues to consider.

Case 1: References to argument data structures
The undo message arguments may contain a reference to a data structure (like a string) as in the following method:

```
- setStringValue:(const char *)value
{
   // Register self and the '-setStringValue:' method
   // with a copy of the old string value
   [[undoManager setUndoTarget:self]
      setStringValue:NXCopyStringBuffer(string)];

   // Free old string, set the new string value and return
   free(string); string = NXCopyStringBuffer(value);
   return self;
}
```

I address this issue with an option to free arguments of registered messagesÐ **freeUndoArgs:(BOOL)***value*. The **setStringValue:** method now becomes this:

```
- setStringValue:(const char *)value
{
   // Register self, the '-setStringValue:' method and a copy
   // of the old string value with the freeArgs option
   [[undoManager setUndoTarget:self] freeUndoArgs];
   [undoManager setStringValue:NXCopyStringBuffer(string)];
```

```
   // Set the new string value and return
   string = NXCopyStringBuffer(value);
   return self;
}
```

Thus the UndoManager stores a flag in the UndoRecord specifying whether to free arguments
of registered methods if the record falls off the end of the list. Using the Objective C run-time function **method_getArgumentInfo()**, the UndoManager can determine the argument types and free them on a case-by-case basis if the type is a reference like string or object.

As a convenience, the method **copyUndoArgs** copies pointer arguments that are passed to the UndoManager. It's assumed that the UndoManager will then be responsible for freeing them.

Case 2: Unneeded objects as receivers
The receiver of the undo message may be an object that's no longer needed after it falls off the end of the undoList. For instance, here's a case of object removal:

```
- removeSelf
{
   // Register '-addSelf' method with the UndoManager
   [[undoManager setUndoTarget:self] addSelf];

   // Remove self from the global list and return
   [globalList removeObject:self];
   return self;
}
```

This scenario is addressed with an option to free the target of the Undo recordÐ

freeUndoTarget:(BOOL)*value*. The **removeSelf** method now becomes this:

```
- removeSelf
{
   // Register '-addSelf' method with the UndoManager with
   // the freeTarget option
   [undoManager freeUndoTarget:YES];
   [[undoManager setUndoTarget:self] addSelf];

   // Remove self from the global list and return
   [globalList removeObject:self];
   return self;
}
```

REDO

So when a user chooses the Undo command from a menu and the UndoManager dispatches a method like **setColor:** to the appropriate target, what happens to the message that's registered from the **setColor:** method during the undo? The obvious thing that should happen is that the UndoManager should save these records in another list, a redoList. Then a **redo:** method could dispatch the last record in the redoList. Redo comes free with undo!

One implementation detail: The redoList is automatically truncated when a new record is added to the undoList, thus it's valid only when the last action was the **undo**: methodÐas it should be.

MULTIPLE RECORDS IN ONE UNDOABLE EVENT

Thus far I have hidden a minor point of complexityÐthe scenario where multiple UndoRecords or Objective C messages make up one undoable event. This would be the

case, for instance, where you are setting an attribute of a list of objects and undo should revert the changes for the whole list:

```
- setColorOfList:(NXColor)color
{
   // Set the new color for every object in the list
   [objectList makeObjectsPerform:@selector(setColor:)
      withColorValue:color];
   return self;
}
```

This problem is addressed by making the undoList not just a list of UndoRecords but a list of lists, which in turn may contain several UndoRecords for one undoable action. The UndoManager
can be told when to collate UndoRecords with the methods **beginUndoRecordGrouping** and **endUndoRecordGrouping**. Our undoable list method now becomes this:

```
- setColorOfList:(NXColor)color
{
   // Set the new color for every object in the list with calls
   // to the UndoManager to collate the changes into one event
   [undoManager beginUndoRecordGrouping];
   [objectList makeObjectsPerform:@selector(setColor:)
      withColorValue:color];
   [undoManager endUndoRecordGrouping];
   return self;
}
```

Individual UndoRecords that are received without the [**begin**/**end**]**RecordGrouping** pair are

assumed to reside in their own group.

SKIPPING INTERMEDIATE ACTIONS

The final and most complex detail of this UndoManager is how to handle the scenario in which intermediate changes occur and only the final result should be undoable, like in a mouse loop. Using the UndoManager as described thus far in a mouse loop would record every discrete change. While this might provide for interesting animation while holding down the Undo menu item, it would prove tedious and waste memory.

The solution is to prevent the UndoManager from registering extraneous events. This is done with the **disableUndoRegistration** and **reenableUndoRegistration** methods. A mouse loop might be implemented in the following fashion:

```
- mouseDown:(NXEvent *)event
{
   // Get the initial value, set it and disable further undo events
   float int = [self getNewValueFromEvent:event];
   [object setValue:value];
   [undoManager disableUndoRegistration];

   // Get next event until mouse up and update value for object
   while(event =[NXApp getNextEvent] && event->type != NX_MOUSEUP) {
      float int = [self getNewValueFromEvent:event];
      [object setValue:value];
   }

   // Reenable UndoManager
   [undoManager reenableUndoRegistration];
   return self;
}
```

UNDOMANAGER GLOBAL VARIABLE?

The previous examples all assume the existence of a global variable named undoManager. In my applications, I allocate an UndoManager for each document and install it into the undoManager global when that document becomes main.

An alternative approach would be to add an application method like the **mainWindow** method:

```
- undoManager { return [[[self mainWindow] delegate] undoManager]; }
```

Thus, all of the above calls to UndoManager could be replaced with **[NXApp undoManager]**.

IMPLEMENTING ANIMATION

By now you are perhaps wondering if the word *animation* was just added to the title to get you to read this article.

The AnimationManager that's available with the UndoManager MiniExample uses the same mechanism for event registering as the UndoManager. However, along with each target-action-arguments record is stored a time stamp set in the AnimationPanel.

When the user clicks the play button on the panel, the animationList is traversed in order of time and registered messages are sent at intermediate times with interpolated values. The arguments are interpolated based on their types, as given by **method_getArgumentInfo()**. Thus if you set the color of an object to red at time 0 and blue at time 10, pressing play would animate the object from red to blue over an interval of 10 seconds.

While this is probably not the most efficient way to do animation, it provides a thorough authoring mechanism for output such as a NEXTIME movie.

PERFECT FOR ALL OCCASIONS!

Ok, so it sounds like I think I've thought of everything. In fact, I have used this object to implement Undo successfully in a major Draw-style application. Compared with the undo strategy used in the NextDeveloper example, Draw, UndoManager seems to require about one-tenth
the number of lines of code. However, there are some caveats that I haven't addressed with this implementation of UndoManager, for instance:

- **forward::** won't be called for messages that the UndoManager actually implements for its own purposes; these include **undo**, **setUndoTarget:**, and others.
- **forward::** can't be used to forward messages with variable number of arguments (varargs).
- There's no mechanism to have the UndoManager selectively free particular arguments when there's more than one in a method call.

If you have feedback on UndoManager, let me know.

Jeff Martin is Director of Software Engineering (and his staff) at Bozell, Inc. He's working on information and graphics applications in the advertisement industry. You can reach him by e-mail at **jmartin@bozell.com**.

Next Article NeXTanswer #1997 **appDidInit:**
Previous article NeXTanswer #2003 **Sneak Preview: The New Foundation Kit**
Table of contents http://www.next.com/HotNews/Journal/NXapp/Summer1994/ContentsSummer1994.html