Undo

Statement of Purpose

The purpose of this chapter is to explain how the undo feature was added to the Draw example application. Our goal is to enable you, an experienced NeXT application developer, to use the ideas presented here to add Undo to your own application. We've designed the Undo code so that the parts not directly concerned with Draw can be easily incorporated into your application.

What is Undo?

Undo enables a user to reverse or rollback the effects of previous and potentially destructive operations. This feature

is most often used to undo an unintended or unexpected action, but it also lets users experiment with differentRWSmands to see how they work. Users can also re-issue an action that was previously undone. This is called ªredo.º

The most obvious manifestation of undo capability in Draw are two new menu items in the Edit menu. These menu items contain the name of actions that can be undone and redone. In this case, actions are things like moving a Graphic, deleting a Graphic, or creating a new Graphic. In the version of Draw that we've compiled for you, two menu items implement multiple-undo. Later on we'll show you how to easily implement single-level undo.

Change Objects

Before we go on, we should formalize the notion of a user action. There are many different kinds of user actions, but we're only interested in the ones that cause the state of a document or an important part of the application to change. If your application uses documents like Draw, then any operation which would normally cause the document to become ªdirtyº should be undoable. Even if your application doesn't use documents, you can still make the editing in your text fields undoable.

From now on, instead of talking about user actions, we'll refer to undoable user actions as ªchanges.º In fact, there's a class called Change that is used to represent changes. Each time the user does something that's undoable, the application will create an instance of a subclass of Change, which we'll call a

ªchange object.º

Each change object encapsulates all the information necessary to undo and redo its corresponding user action. A simple example is a change that represents a modification to the floating point value of a control. The change object for this action would need to record which view got modified, the value before the change, and the new value after the change. Undoing the change is a simple matter of copying the old value back into the control, while undoing the undo (redo) requires that you re-copy the new value into the control.

Implementing a Change Class

To see how this works, lets implement a simple Change class called FloatValueChange. Here's the interface:

```
@interface FloatValueChange : Change
{
   id myView;
   float oldValue;
   float newValue;
}

- initView:changedView;
- saveBeforeChange;
- saveAfterChange;
- undoChange;
- redoChange;
- (const char *)changeName;
RWTnd
```

As described above, the instance variables record the view which will be modified, the original value and the new value. The **initView:** method is the designated initializer for our

class, and the following four methods override standard methods found in the Change class. The last method returns a name string that appears in the undo/redo menu items.

The **saveBeforeChange** method is used to set the oldValue variable, while **saveAfterChange** sets the newValue. When the user wants to undo this operation, **undoChange** will be called to restore the oldValue. Similarly, **redoChange** copies newValue into myView. Here are the implementations:

```
@implementation FloatValueChange

- initView:changedView
{
   [super init];
   myView = changedView;
   return self;
}

- saveBeforeChange
{
   oldValue = [myView floatValue];
   return self;
}

- saveAfterChange
{
   newValue = [myView floatValue];
   return self;
}

- undoChange
{
   [myView setFloatValue:oldValue];
   return self;
}
```

```
- redoChange
{
   [myView setFloatValue:newValue];
   return self;
}

- (const char *)changeName
{
   return("Float");
}

@end
```

All change classes follow the same pattern as the simple one we just created. The change object is responsible for saving the state of the document, view, or whatever object is about to be modified, before and after the modification. It also needs to be able to restore the state to the way it was either before or after the modification.

You might have noticed that FloatValueChange doesn't know what the actual change is. The reason for this is that if the change was a complicated calculation, it could be too expensive or even impossible to duplicate the same calculation twice. So, in general, change objects have no knowledge about how changes are made in the first place, but do understand how to save and restore state information.

Using Change Objects

Obviously, things do change in a running application, so let's examine how modifications are made using change objects. The only method of our undoable control that we need to modify is the one that sets the floatValue. Here it is:

```
@implRWUtation MyUndoControl

- setFloatValue:(float)value
{
   id change;

   change = [[FloatValueChange alloc] initView:self];
   [change startChange];
      floatValue = value;
   [change endChange];

   return self;
}

@end
```

When **setFloatValue:** is called, we know that some other part of the application wants to update the value of the control. The implementation above first allocates a blank change object and then initializes it. The call to **startChange** lets the change object know that the control is about to modify itself. The call to **startChange** will eventually result in a call to **saveBeforeChange**. We didn't have to implement **startChange** in our change class above because it was inherited from the generic Change class.

The next step is to update the internal data structures, with an assignment statement in this case. Finally, we let the change know that we're done by calling **endChange** which ends up calling **saveAfterChange**. This is the basic pattern for any modification to a data structure that should be undoable. Simply create an instance of the appropriate kind of change object and give it control before and after the modification is to be made.

You can write your own classes to know about change objects from the start, but it is often more convenient to create a subclass that adds the change object code. This makes it very easy to add undo functionality to an application that already exists, because you only have to think about undo when everything else already works.

Change Manager

Change objects do most of the work for you in terms of implementing undo. However, there's another part to the story. Whenever the **startChange** method of a change object is called, a search is made up the responder chain to find the nearest change manager.

A change manager is an object that collects the individual change objects and makes them available to the user via the undo/redo menu items. The change manager is also responsible for freeing change objects when they're no longer needed.

As an application runs, its change managers wait for changes to be passed to them via the responder chain. Typically, a view deep in the view hierarchy for a window will create a change objRWVand then call **startChange**. The change object then broadcasts the **changeInProgress:** method on the responder chain. The search up the chain eventually reaches a change manager which replies with a **saveBeforeChange** message.

In document oriented applications, like Draw, it is very easy

to derive your document class from the ChangeManager class. Since document objects are typically installed as the delegate of their window, the ChangeManager will govern all changes that occur within that particular document.

If you would rather implement application-wide undo, simply install a ChangeManager as the delegate of your application, so that all change objects are governed by the same ChangeManager. You can also add ChangeManagers in other places in the responder chain if you need to. However, it might be difficult to determine which ChangeManager should control the undo and redo menu items.

The Undo/Redo Menu Items

The ChangeManager class implements three target-action methods that can be connected to menu items. The first, **undoOrRedoChange:** implements single-level undo. This means that only the last change will be undoable, and after it is undone, the menu shows Redo with the same change. For most applications, its just as easy to implement multiple-undo as it is single-undo.

You might consider using single-level undo if it greatly simplifies the user interface of your application. Also, if you choose not to make the creation and deletion of objects undoable, then you should consider using single-level undo. The reason for this is if you try to redo a modification to an object that doesn't exist (because it couldn't be re-created), either your application or the user could become very confused.

The other two methods, **undoChange:** and **redoChange:** work as a pair. Together these implement multiple-undo. This means that every change going back in time is either undoable or redoable, and there are separate menu items for undo and redo. Connect the undo menu item to **undoChange:** and the redo menu item to **redoChange:**.

Multiple-undo is much nicer for the user, and you should implement it if you can. You'll need to make the creation and deletion of objects undoable for the reasons mentioned above. You should also make sure thaRWWne of your change objects depend on global variables that might be modified between the time the change object was created than the time the user wants to undo or redo a change.

The file ChangeManager.m defines a constant N_LEVEL_UNDO which tells the ChangeManager how many levels of changes to keep track of. To get single-level undo simply set this constant to 1. For multiple-undo set it to any number you like, but give some thought to how large your change objects are likely to be and how much memory you can afford to spend on your undo history.

Updating the Menu Items

The ChangeManager class supports the **validateCommand:** method to automatically update the undo menu items after each change. This method is passed the id of the menu item to be validate. It examines the action field of the menu item to determine which menu item is being validated and will update the title of the menu item to reflect the name of the change to be undone or redone.

If you want to use **validateCommand:** then you'll have to use **setUpdateAction:forMenu:** in the MenuCell class to cause **validateCommand:** to be called when the menu is updated. Draw uses this technique for all menus.

The title of the menu cells are calculated from the **changeName** method of the change objects. The ChangeManager prepends either ªUndoº or ªRedoº as appropriate.

You should call `[NXApp setAutoupdate: YES]` to make sure that the undo menu items reflect the name of the last change after every event. This is especially important if you implement document-level undo. In this case, the undo menu items need to be updated whenever the user brings a new document window to the top.

Making your Application Undoable

Once you understand how the undo mechanism works, it's straightforward to make your application undoable. Here are the steps involved:

1) Examine your application and determine which modifications should be undoable. Then create your subclasses of Change to represent these changes.

2) Decide where your ChangeManagers should be located. For document-level undo, make them delegates of your document objects or derive your document class from ChangeManager. For application-wide undo, put a

ChangeManager behind the application objRWX The important thing is to make sure each ChangeManager is located on the responder chain above any views where change objects will be created.

3) Modify your existing code to create change objects for each user action to be undoable. The easiest way to do this may be to create an undoable subclass of each view that causes changes, like the UndoText subclass of Text in the Draw example. Then you can simply override the methods that update data structures to be like **setFloatValue:** above. Another option is to add change code directly to each view class, which is what we did with GraphicView in the Draw example.

4) Decide whether you want single-level undo or multiple-undo. For single-level, add one new menu item and connect it to your ChangeManager with the **undoOrRedoChange:** method. Do this in Interface Builder. If you want multiple-undo, create two new menu items that are connected to the **undoChange:** and **redoChange:** methods. Make sure that the update actions of these menu items are set to **validateCommand:**.

5) Make sure that the Change and ChangeManager classes along with all your new change classes are linked into the application. After you recompile, you application will have undo!

UndoText

To further simplify your life, we have created a subclass of

Text called UndoText. This class takes care of the chore of making text editing undoable. To use it, simply use an UndoText object where you would normally use a standard Text object. In the presence of a ChangeManager, you'll get undoable text. If there isn't a ChangeManager above UndoText on the responder chain, it will work just like the normal Text object. The UndoText class in Draw overrides selected Text object methods, making all the Draw application interactions with the Text object undoable. If your application uses other features of the Text object you may need to override a few more methods in UndoText.