Reusable Hierarchical Command Objects

Brad A. Myers
Human Computer Interaction Institute
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213
bam@cs.cmu.edu
http://www.cs.cmu.edu/~bam

David S. Kosbie
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052
dkosbie@microsoft.com

ABSTRACT

The Amulet user interface development environment uses hierarchical command objects to support the creation of highly-interactive graphical user interfaces. When input arrives or a widget is operated by the user, instead of invoking a call-back procedure as in most other toolkits, Amulet allocates a command object and calls its DO method. Unlike previous uses of command objects, Amulet organizes the commands into a hierarchy, so that low-level operations like dragging or selection invoke low-level commands, which in turn might invoke widget-level commands, which invoke high-level, application-specific commands, and so on. The top-level commands correspond to semantic actions of the program. The result is better modularization because different levels of the user interface are independent, and better code reuse because the lower-level commands, and even many high-level commands such as cut, copy, paste, text edit, and change-color, can be reused from the library. Furthermore, the commands in Amulet support a new form of Undo, where the user can select any previous operation and selectively undo it, repeat it on the same objects, or repeat it on new objects. In addition, operations like scrolling and selections can be undone or repeated, which can be very useful. Thus, the command objects in Amulet make it easier for developers by providing more reusable components, while at the same time providing new capabilities for users.

KEYWORDS: Amulet, User Interface Development Environment, Toolkits, Command Objects, Undo, Redo.

INTRODUCTION

One of the primary and oldest software engineering principles is to use a "layered architecture," where each layer "is a useful subset of the system" [9]. It would seem that user interface software would naturally decompose into these layers. For example, deleting a file in the finder might decompose into a delete operation, a drag-icon operation, and a low-level object-follows-mouse operation.

CHI 96 Vancouver, BC Canada
© 1996 ACM 0-89791-777-4/96/04..$3.50

Unfortunately, today's toolkits and user interface environments do not support layered software design well, especially for the handling of the users' inputs. Typically, the window manager sends the hardware-level mouse and keyboard events, which the application code directly handles. When using widgets like buttons and menus, the programmer does not have to deal with the low-level events, but programmers still find that the "call-back procedures" invoked by the widgets often become an unstructured and unlayered maze of code. Furthermore, the widgets' implementations are not layered: they directly handle the hardware events. Other toolkits have tried to increase the modularity and reuse by providing a *single* layer. For example, Garnet [6] provides Interactors to handle low-level input, Gina [1] and MacApp [12] provide a single "command" layer, Chiron-1 [10] has the "Artist" layer, and the Macintosh provides the "Apple Events" layer. In all of these systems, the result is that programmers find it difficult to modularize their implementation, because the code must be "factored" into exactly two layers. Since most of the behavior is implemented in application-specific code, there is less reuse of code.

To address these problems, Amulet provides *hierarchical command objects*, where each application-level operation is typically implemented using a number of layers of command objects. This is based on the PhD research on "hierarchical events" by the second author [3, 4, 5]. Instead of invoking a call-back procedure, all interaction techniques and widgets allocate a command object and call its DO method. The DO method will take some action, store any data needed for UNDO, and then invoke the DO method of a higher-level command. For example, when the user selects a menu item to delete a graphical object, the mouse events are first passed to a general-purpose "Choice-Interactor" object [6] which can handle any kind of selection. The DO method in the command object associated with that Interactor calls the command object associated with the menu, which knows how to update the feedback to show which menu item is selected. This command object in turn calls the higher-level Cut command, which removes the object from the window and stores it in the invisible clipboard, as well as in the command object to support undo. Another use for hierarchical commands supports modal dialog boxes. The widgets of a dialog box typically should *not* cause any effect until the OK button is selected. Amulet groups all the commands from the dialog boxes as children of the dialog box's high-level command.

The hierarchical nature of Amulet's commands promotes reuse, since the various layers of Amulet provide libraries of commands that can be used by applications, usually *without change*. For example, the basic low-level text field editing, mouse-based selection, and mouse-based moving and growing of objects are supported by built-in commands. These command objects also support undoing of the operations. Higher-level commands, such as cut, copy, paste, delete, object creation, object property changing (such as changing a color or line-style), are also supplied. Unlike some systems like MacApp, commands in Amulet are used for all application-specific behaviors, including directly manipulating objects. These custom behaviors can often be achieved by simply linking a new command as the parent of an existing command, and then the system will perform (and undo) both. The custom command does not have to repeat any of the code in the lower-level commands.

In addition to increasing the modularity and reuse of code, Amulet's command objects also support a powerful undo mechanism. One of the most difficult tasks when programming an interactive application is providing *undo* for every operation. Thus, while most commercial applications provide Undo, very few research ones do. Because supporting user interface researchers is an important goal of Amulet, the undo mechanism is easy-to-use, flexible and entirely replaceable. So far, we have developed three different undo mechanisms. One is a Macintosh-like single undo, and another supports conventional multiple undo, where the user can undo all operations in sequence. The third form is an innovative *selective undo* mechanism that allows the user to select any previous operation and undo it, as in Gina [1]. Unlike Gina, Amulet also allows the user to *repeat* any previous operation, and to repeat previous operation on a different set of objects. This allows the user to express with a single command, "change the color of *these* objects to be the same as the color that I set *that* object to."

This new undo mechanism also allows the user to easily undo or repeat supporting operations like selections and scrolling. It is common that setting up a complex selection will take a number of steps. The user might accidently clear the selection with a single click in the wrong place. In other systems, the user then needs to start over, but in Amulet, the user can simply undo the last incorrect selection action. Undoing and repeating scrolling is useful when the user wants to repeatedly switch among two or more specific locations in a document. By combining this feature with Amulet's general-purpose facility that allows any command to be marked and executed with a single keystroke, the user can easily set up "bookmarks" in any application, and even in scrollable file lists.

Another innovation in Amulet's undo mechanism is that when a command operates on multiple objects, users can undo the command on a *subset* of the objects. For example, if the user selects a number of objects and then deletes them with a single command, the command can be expanded in the undo dialog box and the user can specifically undo the delete of a single object and leave the rest deleted.

The hierarchical command objects in Amulet also support a number of other important aspects of UI development:

- **Enabling:** A standard field of the command objects controls enabling and disabling of commands (and whether the associated widget is grayed out).
- **Help:** Another field supplies a short help string that can be used for the "help line" in Microsoft Windows or for the "balloon help" on the Macintosh. There is also a pointer that can be used when longer help is needed, which will typically bring up the help dialog box.
- **External Control:** The command objects also allow a program to control another program as if it was the user (a program can easily "push the buttons" of another without requiring any changes in the target application). Since the command objects have a simple, fixed protocol, a program simply sets the `VALUE` slot of the desired command object, and calls its `DO` method.
- **Macros and Programming by Demonstration:** In the future, command objects will be used to create a flexible macro facility (by having the recorder save the commands and call the `DO` methods) [5]. Since this will be supported at the system level, applications will not have to invent their own independent macro mechanisms.
- **Analyzing User Interfaces:** We expect the command objects to be useful for transcribing and analyzing the user's actions, since the hierarchical decomposition of the commands seems to map closely to users' task decomposition.

RELATED WORK

The Amulet system draws much inspiration from the Garnet system [7]. In particular, the low-level input event handling in Amulet uses Interactor objects based on the design from Garnet [6]. However, Garnet did not support any form of command objects, undo, or hierarchical layers for behaviors.

Using command objects to support undo was introduced in MacApp [12] and has been used in many systems including InterViews [11] and Gina [1]. All of these use a similar single-level command architecture. Katie introduced the idea of hierarchical events and explored some implementation issues [3, 4, 5]. Katie also proposed mechanisms for using hierarchical events for scripting and undo. Amulet is the first system to fully explore hierarchical command objects as a way to support large-scale applications, modularity, reuse, and undo.

There is a long history of research into various new undo mechanisms, and Amulet is specifically designed to allow new mechanisms to be explored. Berlage [1] has a good survey of previous undo mechanisms. The selective undo mechanism described here is closest to the Gina mechanism [1], but adds the ability to repeat previous commands, even on new objects. Furthermore, Gina did not support undoing parts of commands or undoing selections and scrolling.

COMMAND OBJECTS

A command object is defined for each operation in the system. Many command objects are built into the system, and

others are created by application programmers. For example, there are built-in command objects for dragging graphical objects, changing objects' colors, and copying objects. Typically, each menu item and widget has an associated command object. Because command objects can be hierarchical, composite objects like menus and menubars can have a separate command object for each item (e.g., a separate command for cut, copy and paste). Alternatively, there might be a single command for an entire menu where the VALUE of the command is determined by the menu choice. For example, in a menu for picking a font, the change-font command can be attached to the menu itself and each menu item can simply be a font value.

Command objects can be attached to widgets either by writing code or by using an interactive tool like an Interface Builder, which might display a menu of built-in and programmer-defined commands. The command object associated with a widget can even be computed by a *constraint*, which is a function that recomputes the value whenever needed. This makes it easy to support "menubar sharing" where the global menubar items change based on which window or sub-window is active.

Slots of Command Objects

Instance variables of Amulet objects, called *slots*, can be dynamically added and removed from any object. Slots that are not set locally are inherited from prototypes. Methods are implemented as function pointers stored in slots. Command objects in Amulet have a number of default slots and methods. The LABEL slot of a command object holds a string or object that will typically be displayed in the widget. For example, menus, buttons and text input fields use the value of the LABEL slot as their label. Because Amulet allows slots to contain any type of value, the LABEL slot can contain a graphical object instead of a string. This allows any picture to be a widget's label, not just bitmaps or strings as in most other toolkits. Keeping the label with the command object rather than with the widget ensures that the same operation consistently uses the same name. Instead of a regular value, the LABEL slot can contain a *constraint* that calculates the value. This is very useful when the label should change. For example, implementing a cycle button that alternates between `"turn grid ON"` and `"turn grid OFF"` is trivial.

The ACTIVE slot of a command object controls whether the associated widget, Interactor, or menu item should be enabled or not (grayed out). Typically, the ACTIVE slot contains a constraint. For example, the Cut command object's ACTIVE slot contains a constraint that returns false when no objects are selected. The VISIBLE slot of a command object can be used to control widgets, dialog boxes and windows that appear and disappear. If the VISIBLE slot is true, the associated graphical object is visible, and if false, then it is invisible. Using the VISIBLE slot of the attached command object to control pop-up dialogs provides a nice, uniform interface.

Command objects can also contain HELP and SHORT_HELP slots to support the help-line and full help text for the command. The help string in a command can be computed by a constraint. This makes it easy, for example, to change the string from telling what the command does to telling why it is not available based on whether the ACTIVE slot is true or false.

Since Amulet allows a program to dynamically add slots to objects and since a slot can hold any type of data, command objects typically store their current and previous states as slot values. Four standard slots are defined, but custom command objects can add other slots as necessary. The standard slots are the VALUE slot for the current value, the OLD_VALUE slot for the previous value, the OBJECT_MODIFIED slot for the object or list of objects modified by this command, and the OLD_OWNER slot for the widget or Interactor that invoked the command. By using these standard slots for holding the data of the command, the standard undo mechanisms can provide default ways of checking and displaying the commands, as described below. These slots also form the interface between lower-level and higher-level command objects in the hierarchy. For example, the low-level Interactor which moves objects is used in the scroll bar widget to let the user drag the indicator. When the user drags the indicator, the DO method in the command in the Interactor sets its VALUE to the current position. The scroll bar command's DO method then uses this value to compute the appropriate percent value, which is then set into its own VALUE slot.

Methods in Command Objects

Amulet calls the command's DO method when the command should be invoked. The various kinds of undo mechanisms require additional methods in each command, as described below. The standard single and multiple undo require two methods (for undo and undo-the-undo), and the selective undo mechanism requires six methods. In some situations, even for custom commands, it may not be necessary to use a DO method at all. Instead, constraints can be used. For example, the default change-property command stores the current value of the property in the VALUE slot of the command, and often a constraint can be put into objects to get the current value of this slot. Then, the DO method can be NULL (empty), and the UNDO method will be the default, which simply swaps the values in the VALUE and OLD_VALUE slots of the command. For commands like cut or delete-file, however, both DO and the various UNDO methods are needed.

Command Object Hierarchies

Command objects in Amulet use *four* different hierarchies (see Figures 1 and 2). All Amulet objects, including commands, are in a prototype-instance hierarchy. This is used to control the inheritance of default values for slots and methods, and corresponds to the class hierarchy in C++. In Figure 1, the command in item 1 is an instance of the Cut command, and inherits a number of methods and slots from it. Other methods and slots might have local, application-specific values. The Cut command, in turn, is an instance of the root Command object.

Most objects in Amulet also participate in a part-owner hierarchy. This corresponds to the grouping mechanism in other toolkits. Graphical objects are typically part of

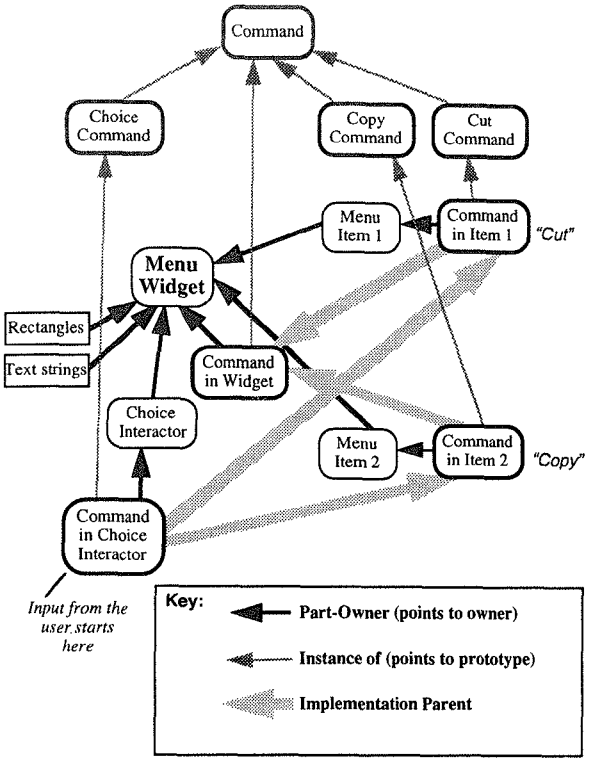

Figure 1:
Three of the hierarchies that command objects use.

groups, which are part of a window, which are part of the screen. Non-graphical objects can also be added as parts. This supports *structural inheritance* because all parts of an object are copied when an object is copied. In Figure 1, the menu widget contains various rectangle and string graphical objects as parts, as well as a Choice-Interactor object, the command object for the widget, and the command objects for the various menu items. When a copy or instance is made of the menu object, Amulet also copies all of these parts. Programs can safely store local values into the command objects since a unique command object will be allocated for each menu and menu item.

The other two hierarchies are specific to command objects, and are not used by other kinds of Amulet objects. The first one supports lower-level commands executing the higher-level commands. For example, in a menu, the low-level command associated with a Choice-Interactor deals with the interim feedback that shows which menu item the mouse is over. When the user releases the mouse over a menu item, the Choice-Interactor's command turns off the interim feedback, and then calls the DO method of the appropriate command associated with that item. If desired, the menu's command object would be invoked next. We call this the "implementation hierarchy," since all the commands work together to implement the operation.

The implementation hierarchy is quite different from the prototype-instance inheritance hierarchy, since in the prototype-instance hierarchy, an instance's DO method *replaces* the parent's DO method. In the implementation hierarchy, however, the child's DO method is run in addition to the parent's DO method (specifically, the child's runs *before* the parent's).

Amulet uses the IMPLEMENTATION_PARENT as follows: when an input event arrives, Amulet finds the Interactor object to handle it. Then, that Interactor's command object's DO method is invoked, followed by its IMPLEMENTATION_PARENT command's DO method. In pseudo-code:

```
cmd = current_interactor.COMMAND;
do {
     cmd.DO_Method();
     undo_handler.Register_Command(cmd);
     cmd = cmd.IMPLEMENTATION_PARENT;
     }
 while (cmd.Valid());
```

Of course, the standard widgets set the IMPLEMENTATION_PARENT slots of internal commands automatically, so the programmer is only responsible for higher-level parts of this hierarchy. For custom widgets and behaviors on application-specific objects, programmers will typically use the lower-level commands from the library and attach their own higher-level commands as the IMPLEMENTATION_PARENTs.

Originally, we thought that the part-owner hierarchy would serve as the implementation hierarchy as well, since in most cases commands in parts would call the commands in the owners. For instance, the command in the Move-Interactor in a scroll bar calls the command in the scroll bar, and the scroll bar is the Move-Interactor's owner. However, there are a number of situations where the part-owner hierarchy does not match the required implementation hierarchy. For example, a single Choice-Interactor is attached to an entire menu widget, whereas there may be command objects associated with each individual menu item (see Figure 1). In this case, the Choice-Interactor's command object *dynamically* computes its implementation hierarchy parent based on where the user points. Another example is the selection handles widget, that allows graphical objects to be selected, moved, and grown, and displays the standard square handles around the objects selected. Since this single widget performs multiple actions, it contains a number of Interactors as parts (each with its own low-level commands), and the widget also contains multiple high-level semantic commands such as "become selected" and "move." The part-owner hierarchy is insufficient to match these up.

Our original design for the implementation hierarchy required that programmers call the correct methods as part of their DO and UNDO code, but the programmers using Amulet found it difficult to know what code was supposed to be called when. Therefore, the current design uses a *declarative* approach where the IMPLEMENTATION_PARENT slot in each command object is set with the parent object. Now, the application's DO and UNDO methods can contain arbitrary code, or even be empty. This seems to be much more intuitive.

By contrast, the programming model introduced in Katie [4] provides for the automatic determination of parent information in events by using a nondeterministic parsing

scheme operating on a declarative specification of events accepted by each event handler.

The final hierarchy used by command objects is for situations where multiple widgets are used to construct a single operation, such as with dialog boxes. So far, every command has been either a top-level command (i.e., like "Cut") that will show up in the undo menu, or else the IMPLEMENTATION_CHILD of a top-level command.[1] We found that these two options were not sufficient for every case, since all commands on the implementation hierarchy are executed at the same time. In particular, dialog boxes are often used to execute a single command, but the dialog box itself is composed of many widgets each of which has its own command. In Figure 2, the Change-Font command should *not* be executed when the font-size widget's commands are executed, but only when the OK or Apply buttons of the dialog box are hit. Thus, the Change-Font command cannot be the IMPLEMENTATION_PARENT of the font-size widget. However, we still would like the font-size command to be associated with the Change-Font command for later undoing or transcription. Therefore, commands can have their DEFERRED_EXECUTION_PARENT slot[2] set with the top level command that will eventually be executed. Unlike the IMPLEMENTATION_PARENT command, the DEFERRED_EXECUTION_PARENT command is *not* executed when the child's is. Tools that help programmers build dialog boxes in Amulet, like Interface Builders, will set the DEFERRED_EXECUTION_PARENT slots appropriately, so we do not expect that most programmers will directly need to deal with this hierarchy.

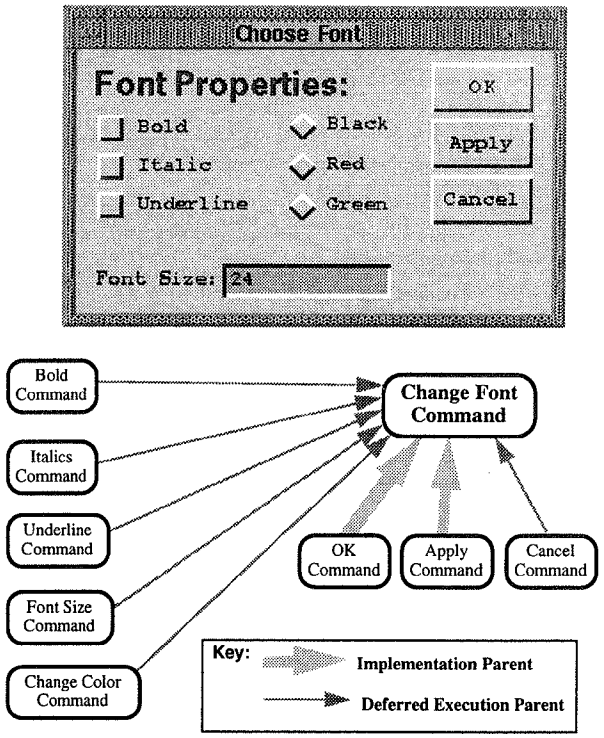

Figure 2:
The Change-Font command is the IMPLEMENTATION_PARENT of the OK and APPLY commands so it will be executed automatically when they are executed, but it is the DEFERRED_EXECUTION_PARENT of the other commands so it will *not* be executed when they are executed, but so the commands will still be linked to the Change Font command.

UNDO AND REPEAT MECHANISMS

User Interface

The command objects themselves have no user interface, since they are used as an execution framework. The general undo mechanism supports a variety of user interfaces, and any Amulet application can supply a custom user interface to the undo mechanisms. This section presents the default user interface for the three undo mechanisms, which we have used to verify that the various options can be presented in an understandable way.

At the top of the Edit pull-down menu in applications is the usual "Undo" option, which undoes the last command. In the Macintosh-like single undo mechanism, this menu item changes to "Redo" after an undo has been performed, which will undo the undo. The usual keyboard accelerator keys can be used for the menu items.

In the multi-level undo mechanism, there are two undo menu items: "Undo" which undoes the last operation that has not been undone, and "Redo" which undoes the previous Undo. The "Undo" option is always available unless the user has undone all the operations back to the initial state. The "Redo" option is only available if the previous operation was an Undo, since this undo model does not support trees of commands, just a linear list.

With the selective undo mechanism, there are three undo menu items in the Edit menu: "Undo," "Redo" or "Repeat," and "Selective Undo/Redo/Repeat...". "Undo" and "Redo" operate the same as for the multi-level undo mechanism. "Repeat" is a quick way to selectively repeat the previous command. "Repeat" shares the same menu item as "Redo" to save space, as in Microsoft Word version 6. The final option brings up the dialog box for selective undo shown in Figure 3.

The undo dialog box lists all the commands, and has buttons for the various undo and repeat actions. The dialog box is not modal, and if it is left visible, the user can see each new command added to the top of the list. The commands are described by their type, the name or a picture of the object modified (if any), and the value of the command (such as the resulting color for a change-color command). If the names of the object is not sufficiently meaningful, the object associated with a command can be made to flash. The user can easily undo or repeat the command to see if it is the desired operation. If the selected command cannot be undone or repeated, the appropriate action buttons are grayed out. This would typically happen if the objects the command affects are no longer valid, for example if the objects have been deleted.

[1]The IMPLEMENTATION_CHILDREN of a command C are all the commands whose IMPLEMENTATION_PARENT is C.

[2]Better suggestions for the name would be appreciated!

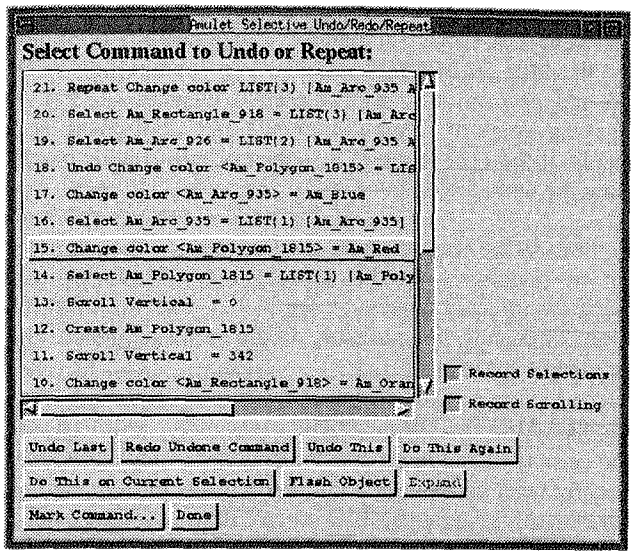

Figure 3:
The dialog box currently used for selective undo, redo and repeat. Command 11 has been marked with the F9 accelerator key.

The meaning of selectively undoing a command has been adopted from Gina [1]: the affected values of the objects are returned to the state just before the command was executed. If a command turns an object from blue to yellow, selectively undoing the command will make it blue, no matter what its current color is. Similarly, selectively repeating the command will turn the object yellow again. Selectively repeating a command on new objects will do the same operation to the new objects. In this case, the new objects would turn yellow. The selective operations also generate a new command object which is put at the top of the command list, and this new command can then be undone using the normal mechanisms. The name of the new command is prefixed with "Undo" or "Repeat," as shown by commands 18 and 21 in Figure 3.

Selective repeating of some commands might mean different things. For example, we decided that repeating a "move" command should put the object in the same place as the original move. However, it might alternatively mean to move by the same *relative* distance. User testing will be required to see which meaning is more often desired, or if users would prefer to have a choice of both options.

The check boxes at the right of the dialog box in Figure 3 control whether scrolling and selections should be queued for undoing. If so, they are added to the command list like regular commands.

The Expand button in the dialog box expands an operation that affects multiple objects into a set of commands, so they can individually be undone or repeated. The selected command can be *marked* and given a user-defined name and keyboard accelerator that will selectively perform the command again on the same object or the currently-selected objects. This can be used, for example, to create "bookmarks" for scrolling to specific places in a document, or to create a command to set any object to a particular custom color. Another use would be to mark the final command which selects a set of objects so that various operations can be applied to them without having to laboriously re-select them or place them into a group. Eventually, marking will be augmented by a full macro facility.

Implementation of Undo

Amulet is designed to be flexible to support user interface researchers. In particular, the undo mechanism is completely replaceable. An undo-handler object is attached to each window, and Amulet calls a standard method in it to process each command executed. Programmers can use one of the three undo-handlers supplied in the library, or they can write their own. Amulet simply requires that each undo-handler support the method REGISTER_COMMAND which takes a command object, and saves it for later undoing.

The undo-handler's REGISTER_COMMAND method checks to see if a command has an IMPLEMENTATION_PARENT or DEFERRED_EXECUTION_PARENT, and if not, then it makes a copy of the command and all of its children. If a command does have an IMPLEMENTATION_PARENT or DEFERRED_EXECUTION_PARENT, the parent command will eventually be executed, and it will then be queued for undo along with its parent, so there is no need for the undo-handler to queue it.

When a single command is undone or repeated, all of its IMPLEMENTATION_CHILDREN commands have to be undone or repeated also, since they all participated in the operation of the command. Consider a widget to change an object's color in a graphics editor. When the user clicks on it, the DO method of the command in the widget changes the widget to show the current color, and the DO method of the application's Change-Color command (which is the IMPLEMENTATION_PARENT of the widget's command) changes the color of the selected objects. If this command is undone, then *both* the widget and the selected objects should return to their previous states. Therefore, the UNDO method in the command object in the widget *and* the UNDO method in the application command should be executed. This clearly shows the advantages of the hierarchical commands, since in other toolkits, the undo method for the application has to know which widget it was part of and how to reset the widget. In Amulet, however, the application command can be exclusively concerned with updating the application's data structures, and the child command for the widget will update the widget.

It is important that the UNDO methods be executed in the *same* order as the DO methods (from child to parent) since often the higher-level commands compute their values from the values of lower-level commands. For example, since the widget's command's DO and UNDO methods will always be executed first, the methods in the application's Change-Color command do not need to store an old value since they can simply access the current value of the widget's command object.

We have found that the DEFERRED_EXECUTION_CHILDREN of a command do *not* need to be undone when

a command is undone. For example, undoing the font setting command should usually not reset the various widgets in the font dialog box. If necessary, an undo method is free to call the undo methods of its DEFERRED_EXECUTION_CHILDREN, and some future undo mechanisms might even allow the DEFERRED_EXECUTION_CHILDREN commands to be individually undone by the user.

Some commands are not undoable (like Save File) but should still be queued on the undo list to show that they were executed. Other commands are not normally queued on the undo list at all (like scrolling and selection). Commands that are not undoable but should still be queued can simply leave their UNDO_METHOD as empty (NULL). The undo-handlers grey-out commands without an undo method. Commands that are not normally queued for undo are so marked, but they still have UNDO_METHODs, so the user can dynamically decide whether they should be queued for undo, for example by using the check boxes in Figure 3.

When an command object can no longer be repeated or undone, it is destroyed. The single undo mechanism destroys the old command object whenever a new command is executed. The multiple command mechanism currently supports unlimited undo, so command objects are only destroyed if they are undone, and then a new command is executed, since at that point, redo is no longer available.

Implementation of Selective Undo and Repeat

Implementing selective undo and repeat is not much harder than implementing regular undo. All the required information is already stored in the command object. The major new complication is that for selective undo or repeat, the affected object's *current* values must be loaded into the command object, in case the selective undo command itself is undone. To facilitate this, the undo mechanism first makes a copy of the command object to be undone or repeated before calling the selective undo or repeat method. Therefore, the methods are free to reload the OLD_VALUE slot from the current object without affecting the original command. For example, the method that implements all the undo and repeat operations for property changing commands looks like (in pseudo-code):

```
// undo is true when undoing and false when repeating.
// on_new is true when applying this to the new object new_obj.
void property_repeat_undo(Am_Object cmd,
          bool undo, bool on_new,
          Am_Object new_obj) {
 if (on_new) {
   cmd.OBJECT_MODIFIED = new_obj;
   object = new_obj;
   }
 else object = cmd.OBJECT_MODIFIED;
 cur_value = object.property_slot;
 if (undo) new_value = cmd.OLD_VALUE;
 else new_value = cmd.VALUE; //repeat
 object.property_slot = new_value; //Set it
 //now swap old and new values in command
 cmd.VALUE = new_value;
 cmd.OLD_VALUE = cur_value;
}
```

If the command affects a *set* of objects, then the OBJECT_MODIFIED slot simply contains a list of objects, and the OLD_VALUE and VALUE slots contain a list of the old and new values. This makes it easy to support the breaking apart of these commands into individual commands, if the user requests that the command be expanded using the undo dialog box.

Methods in the command object determine if the command can be selectively undone or repeated. These typically check the affected objects to see if they are still valid (but some commands, like deletion, do the opposite and check if the objects are invalid).

STATUS AND FUTURE WORK

The command and undo mechanisms described above have recently been incorporated into the Amulet system, and are available for general use.[3] Our early impressions are that it is not difficult to construct applications using the hierarchical command architecture, supporting selective undo is only slightly harder than regular undo, and both are facilitated by the hierarchical commands. In the future, we hope to gather some data about end user's feelings about the selective undo mechanisms: Is the meaning of selective undo and repeat clear? Can users pick the correct command from the dialog box? Is undoing and repeating of selections and scrolling useful and easy enough to do?

Currently, the undo mechanism only allows the user to select the top-level command for undoing and repeating, and this automatically undoes or repeats all the children commands. It would be interesting to explore whether it is useful for users to be able to undo only the children commands, for example, to re-enter a dialog box or other multi-step action and finish it differently. Given the ease of creating new undo mechanisms in Amulet, it becomes attractive to explore many other forms of Undo. For example, what if the previously undone operations were not flushed when new operations were entered, but instead the system supported multiple paths of execution with the possibilities of skipping around? Then the command history might become a complex tree. Combined with the bookmark facility, this might allow "what if" explorations, where the user can try different sequences of actions, and then use the one that works best (even if it wasn't the last one tried).

A number of researchers have explored undo in the context of multi-user applications, and the selective undo model described here seems ideal for this. The interface could easily allow commands like "Undo *my* last action" versus "Undo *the* last action" by simply marking each command with a user ID.

Another important direction for future work is to support scripting and programming by demonstration [8] using the

[3]The complete Amulet system is available for free by anonymous FTP, including the command and undo model described here. See http://www.cs.cmu.edu/~amulet or contact the first author for more information.

command objects so users can aggregate several actions into a single new command. Scripts can be record the command objects at any level: at the low level input events (left button down at (45,30), move to (54,31), ...), at the intermediate command level (move file_object_461 to (54,31)), or at a higher semantic command level (move_file "commands_paper" from folder "submitted" to folder "CHI96"). Different uses of scripts may require recording at any of these levels and having the semantic level will make scripting more useful than previous systems [5]. It will be interesting to explore how to let the user choose which level is desired. Having scripting at the toolkit level, instead of using a different scripting mechanism for every application, will allow users to transfer their knowledge and to create more elaborate programs that use multiple applications.

Transcripts of the multi-level commands may be useful for analysis. The time spent scrolling, selecting, using help, aborting commands, and undoing, can be clearly differentiated from productive work, and it might be possible to write programs to filter the transcript to look for problem areas. Also, it appears that the command hierarchy matches the NGOMSL method decomposition [2] so model-based evaluation of Amulet interfaces may be possible even before testing with any subjects.

Another direction for future work is to extend the selective undo mechanism to text editing. The main problem for selectively undoing and repeating will be *where* in the document a command refers, given that the text may have been substantially edited since the command was first executed.

CONCLUSIONS

The hierarchical command objects in Amulet make it easier to build interactive applications, because low-level commands can be reused from the library without change, and application-specific properties can be easily expressed with constraints and methods in high-level commands. At the same time, the command objects enable a very flexible Undo mechanism that supports selective undoing and repeating of previous actions, and new undo mechanisms can be easily added. We are very excited about the potential of Amulet's command objects as a foundation for future research and for general user interface development.

ACKNOWLEDGEMENTS

For help with this paper, we would like to thank Alan Ferrency, Rich McDaniel, Brad Vander Zanden, John Pane, and Bernita Myers. The Amulet system is being developed by Brad Myers, Rich McDaniel, Alan Ferrency, Andy Mickish, and Alex Klimovitski.

This research was partially sponsored by NCCOSC under Contract No. N66001-94-C-6037, Arpa Order No. B326, and partially by NSF under grant number IRI-9319969. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Government.

REFERENCES

1. Thomas Berlage. "A Selective Undo Mechanism for Graphical User Interfaces Based on Command Objects". *ACM Transactions on Computer Human Interaction 1* (Sept. 1994), 269-294.

2. Michael D. Byrne, Scott D. Wood, Piyawadee Sukaviriya, James D. Foley and David E. Kieras. Automating Interface Evaluation. Human Factors in Computing Systems, Proceedings SIGCHI'94, Boston, MA, April, 1994, pp. 232-237.

3. David S. Kosbie and Brad A. Myers. A System-Wide Macro Facility Based on Aggregate Events: A Proposal. In Allen Cypher, Ed., *Watch What I Do: Programming by Demonstration*, MIT Press, Cambridge, MA, 1993, pp. 433-444.

4. David Kosbie. *Hierarchical Event Histories in Graphical User Interfaces*. Ph.D. Th., Computer Science Department, Carnegie Mellon University, 1996. In progress.

5. David S. Kosbie and Brad A. Myers. Extending Programming By Demonstration With Hierarchical Event Histories. In Brad Blumenthal, Juri Gornostaev and Claus Unger, Ed., *Human-Computer Interaction: 4th International Conference EWHCI'94, Lecture Notes in Computer Science, Vol. 876,*, Springer-Verlag, Berlin, 1994, pp. 128-139.

6. Brad A. Myers. "A New Model for Handling Input". *ACM Transactions on Information Systems 8*, 3 (July 1990), 289-320.

7. Brad A. Myers, Dario A. Giuse, Roger B. Dannenberg, Brad Vander Zanden, David S. Kosbie, Edward Pervin, Andrew Mickish, and Philippe Marchal. "Garnet: Comprehensive Support for Graphical, Highly-Interactive User Interfaces". *IEEE Computer 23*, 11 (Nov. 1990), 71-85.

8. Brad A. Myers. "Demonstrational Interfaces: A Step Beyond Direct Manipulation". *IEEE Computer 25*, 8 (August 1992), 61-73.

9. David L. Parnas. "Designing Software for Ease of Extension and Constraction". *IEEE Transactions on Software Engineering SE-5*, 2 (March 1979), 128-138.

10. R. Taylor, K. Nies, G. Bolcer, C. MacFarlane, and K. Anderson. "Chiron-1: A Software Architecture for User Interface Development, Maintenance, and Run-Time Support". *ACM Transactions on Computer Human Interaction 2* (June 1995), 105-144.

11. John M. Vlissides and Mark A. Linton. "Unidraw: A Framework for Building Domain-Specific Graphical Editors". *ACM Transactions on Information Systems 8*, 3 (July 1990), 204-236.

12. David Wilson. *Programming with MacApp*. Addison-Wesley Publishing Company, Reading, MA, 1990.