A Command Editor Tool for NEXTSTEP

Final Report
July 1, 1993

Sponsored by
Defense Advanced Research Projects Agency (DOD)
Defense Small Business Innovation Research Program
ARPA Order No. 5916

Issued by U.S. Army Missile Command
under Contract # **DAAH01-93-C-R013**
Effective Date: January 15, 1993
Expiration Date: July 15, 1993

prepared by
Patrick Dean Rusk, Minh Huynh, Greg Burd
Marble Associates, Inc.
38 Edge Hill Road
Waltham, MA 02154
(617) 891-5555

July 1, 1993

A Command Editor Tool for NEXTSTEP

MARBLE ASSOCIATES, INC.

The final report prepared for DARPA under Contract # DAAH01-93-C-R013.

Final Report

1.0 Executive Summary

Marble Associates, Inc., is engaged in the development of customizable menus in NEXTSTEP and the design of customizable menus in the X windowing environment. In this document, we present the final design and implementation details of the *Command Editor* in NEXTSTEP. To reiterate the objectives stated in previous submissions, we strive to do the following in building the *Command Editor*:

- retain all functionalities of the standard NEXTSTEP environment;
- add functionalities in a manner keeping with NEXTSTEP, mimicking the user and developer interaction mechanisms of NEXTSTEP; and
- maintain interoperability among our customized objects and standard system objects.

Our realization of the *Command Editor* for NEXTSTEP adheres closely to the original design presented in Progress Report #1. As suggested in that report, newly discovered challenges necessitate slight departures from the design. Our objective is to provide to the user and the developer the functionality outlined in the Phase I Proposal and the Progress Report #1. This document discusses in detail the challenges we faced, our solutions, and the subsequent impact on the design and functionality of the *Command Editor*. We also address some adjustments to our original targeted functionalities forced by the NEXTSTEP environment.

Though a complete discussion is presented, this document assumes that the reader is familiar with concepts and terminology presented in previous papers submitted under Contract # DAAH01-93-C-R013. For a listing of previous submissions, please refer to Appendix B.

2.0 Background

This section introduces terms and concepts necessary for subsequent discussions. We present a synopsis of user and developer functionalities to recap the concepts described in previous submissions. For a complete functional specification, please refer to *The Command Editor: A Manual for Users and Developers*.

2.1 Menu Types

We have separated the menu systems deployed in the Command Editor into three states. Each state represents a distinct stage in the life cycle of the application menu, from its construction in Interface Builder (IB), through its initial instantiation at runtime, to its modification by the end user (Figure 1).

- The **FullMenu** is the menu structure composed by the developer to contain the complete set of commands available for end user customization.
- The **BasicMenu** is also determined by the developer and represents the initial menu configuration visible at runtime prior to any user customization. The **BasicMenu** is a subset of the **FullMenu**. The developer composes this menu in IB by marking members of the **FullMenu** as "Default."
- The **CustomizedMenu** is the end user modified menu. This menu is a modified **BasicMenu** and can contain **CEMenuCells** from the **FullMenu** not marked "Default" by the developer.

We note that the **BasicMenu** and **CustomizedMenu** are not separate objects; rather, the visible menu is in one of these states. When the application displays the initial menu configuration dictated by the developer, we call the menu structure the **BasicMenu**. When the user alters the application menu, we call the menu structure the **CustomizedMenu**. The distinction is important when we discuss the developer's interactions with the Command Editor in IB and again when we describe the **CEController** actions on startup. Indeed, we will refer to the application menu as the "visible menu" regardless of its configuration (**BasicMenu** or **CustomizedMenu**) only when the distinction is irrelevant.

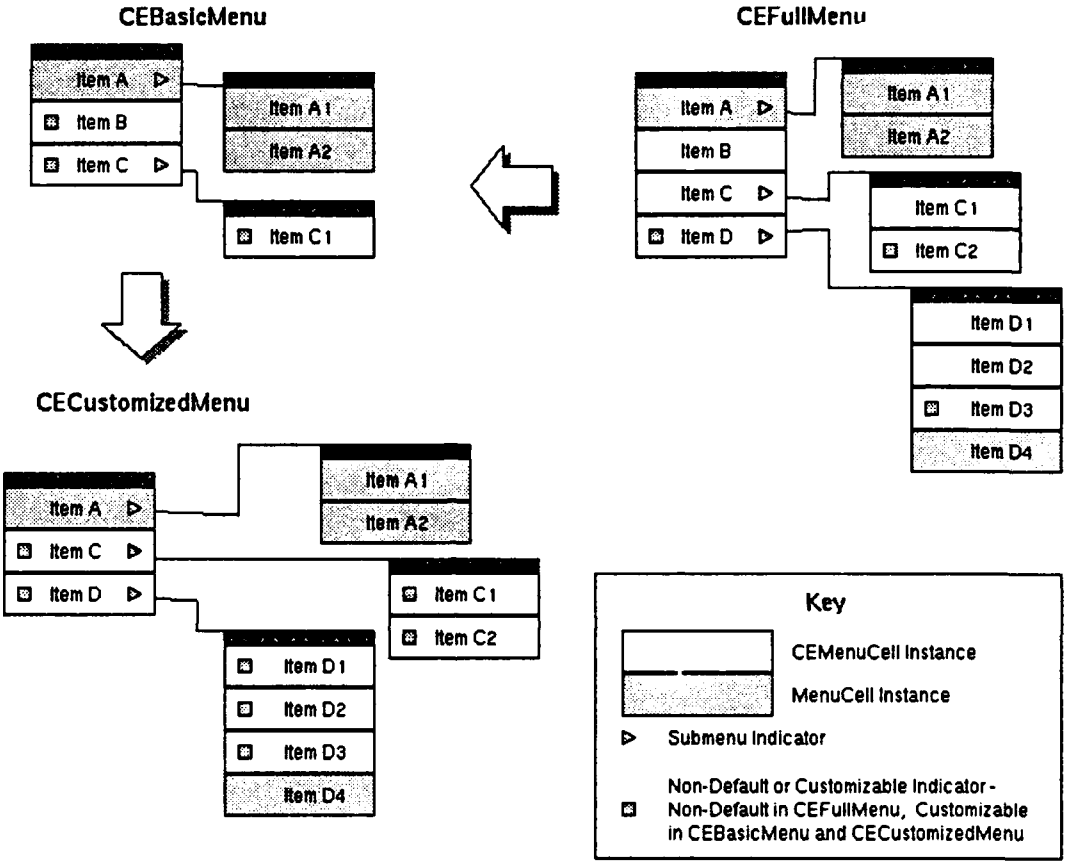
The "invisible" menu, then, is the **FullMenu**. This structure represents the topology of available commands configured by the developer. It is "invisible" in the sense that the end user never directly interacts with this structure as a menu. The user can browse through this topology using the **Menu Editor**, however. The **FullMenu** is stored in the application NIB at the completion of developer configuration in IB. Within the **FullMenu**, certain **CEMenuCells** are marked "Default" by the developer via the **CEMenuCellInspector**.

At runtime, the application will search the user's home directory for a profile (*.applicationName*.ceinfo). If this file is available, the **CEController** loads the customization in this file. The load process instantiates the archived menu structure. The CEController then makes this newly instantiated, customized menu the application menu. If the customization file is not available, the **CEController** constructs the **BasicMenu** by filtering the **FullMenu**.

When the end user chooses the "Configure Menus" option in the "Menu Configuration" submenu, the **Menu Editor** appears and presents the hierarchy of **CEMenuCells**

in the **FullMenu**. The user can then browse and select particular menu items to be added via drag and drop (DND).

FIGURE 1 Menu Types



2.2 Developer Functionality

The developer builds a customizable menu for his application by loading the *Command Editor* palette in IB and dragging our objects from this palette into the application NIB and menu structure (Figure 2). He is responsible for composing the **FullMenu** and **BasicMenu**. Additionally, the developer can incorporate **TAVs** (Tool Acceptor Views) into his application by dragging the **TAV** object from the palette and dropping it in any window in the application NIB. **TAVs** allow the end-user to load often-used menu commands into readily accessible buttons.

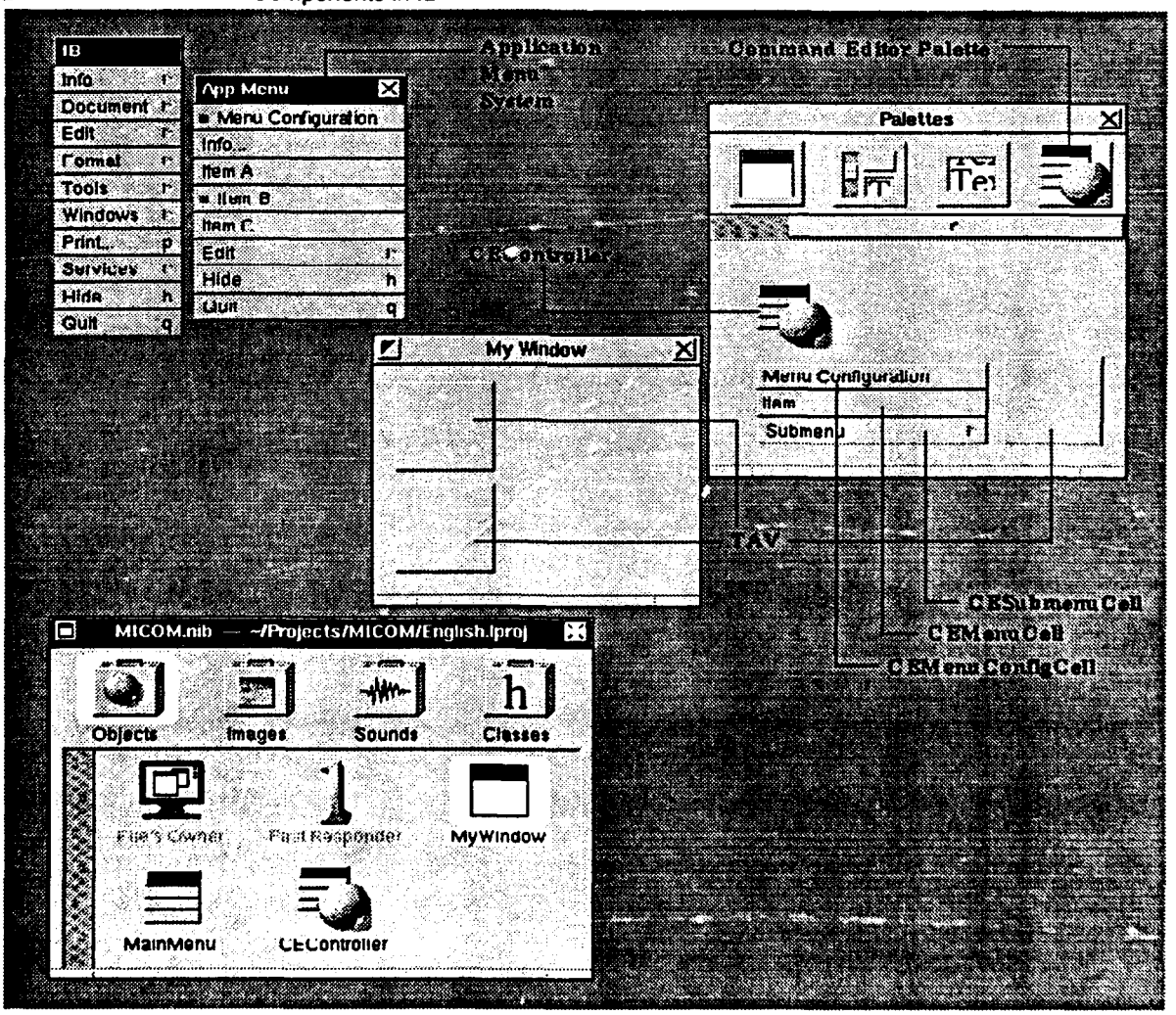
The developer composes the **FullMenu** by dragging **CEMenuCells** or **CESubmenuCells** from our palette onto the application menu. This adds new menu items to the **FullMenu** hierarchy. The developer configures individual items (**CEMenuCells** and **CESubmenuCells**) using the appropriate IB Inspectors provided by our palette.

The developer specifies all of the following for each **CEMenuCell** through the **CEMenuCellInspector**: action, icon, "Default" status, state list, textual description, tool sta-

tus, and disable status. A visual indicator will distinguish "Default" items; a small square notch denotes the exclusion of that item from the **BasicMenu**.

CESubmenuCells allow the developer to add and populate menu curtains. Though we implement the **CESubmenuCells** as a special case of **CEMenuCell**, we will continue using the term **CESubmenuCell** for the sake of clarity. **CESubmenuCells** require the developer to specify the "Default" and "Disable" status via the **CEMenuCellInspector**. We defer the discussion of **CESubmenuCell** implementation to Section 4.1.4. The developer can also specify the border used in drawing a particular **TAV** via the **CETAVInspector**.

FIGURE 2 Components in IB



The **CEMenuConfigCell** represents the "Menu Configuration" submenu that allows the user to customize the application menu. Its submenu items include "Configure Menus," "Save Configuration," "Load Configuration," and "Show/Hide Configurable Cells." Thus, its addition to the menu structure is required if the end-user is to browse

the set of available commands, customize TAVs, or load and save configurations. Without the **CEMenuConfigCell**, the end-user can still Control-drag configurable cells to reorder or delete menu items. The developer simply drags the **CEMenuConfigCell** from the palette and drops it onto the application menu. Only one **CEMenuConfigCell** should be added to an application menu. If more than one is added, all but the first instance in the menu structure are ignored

The developer can incorporate standard, non-customizable MenuCells into the application menu. While these menu items cannot be customized or deleted by the end user, their position in the menu structure can be altered with the addition and deletion of customizable menu items.

2.3 User Functionality

The goal of Marble's *Command Editor* is to provide the functionalities described below. The user can customize various menu characteristics within limits dictated by the developer. Through the "Configure" submenu, the end user can view the set of configurable items in the menu structure, save or load configurations, and customize the menu through the **Menu Editor** (Figure 3).

When requested at runtime, the *Command Editor* classes bear a visual indicator—a small square notch left adjusted in the menu item. These items can be deleted and reordered by Control-dragging them off the menu structure.

The **Menu Editor** allows the user to browse the **FullMenu** and select a customizable item to be added to the application menu or **TAV**. The user can Control-drag a **CEMenuCell** or **CESubmenuCell** from the browser and drop it into the desired position in any menu curtain (Figure 4). Dropping the **CEMenuCell** onto a **TAV** forces the **TAV** to display the icon associated with that **CEMenuCell**. Note that **CESubmenuCells** cannot be dragged onto **TAV**s.

The end user can add submenus by Control-dragging the button from the "Additional Submenu" portion of the **Menu Editor** onto the application menu (Figure 5). The title of this submenu can be entered in the text field immediately below the button. The submenu curtain added will contain an unconfigured item initially (Figure 6). The user can then configure this new menu curtain via DND.

CKEs allow the user to type Command-Key to activate a menu selection without using the mouse. To specify the **CKE** of any configurable menu item, the user enters the new character in the "Command Key Equivalent" field of the **Menu Editor**. The **CKE** will update for items currently displayed in the application menu.

The user can configure a **TAV** by dragging a command from the **Menu Editor** (Figure 7) or visible menu and dropping that command on the desired **TAV**. This command must have its **hasTool** field enabled, indicating a valid image in its **icon** field.

As a final note, we do not allow the user to add menu items to the "Menu Configuration" submenu. The user cannot delete items from this menu curtain nor can he specify CKEs. Indeed, the **Menu Editor** displays the "Menu Configuration" item in "grayed out" fashion, precluding user interaction with this item.

FIGURE 3 The Menu Editor

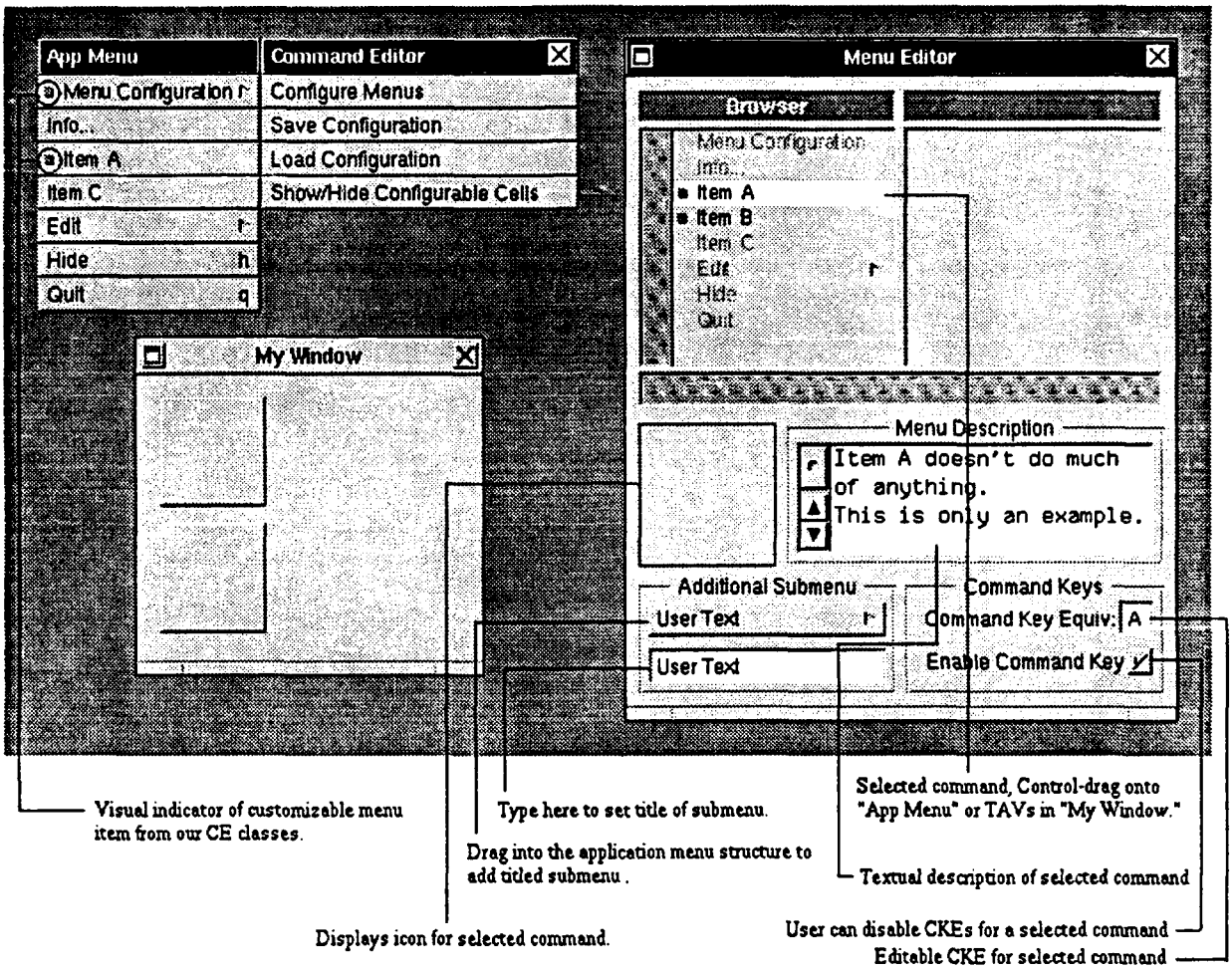


FIGURE 4 Menu Item Addition

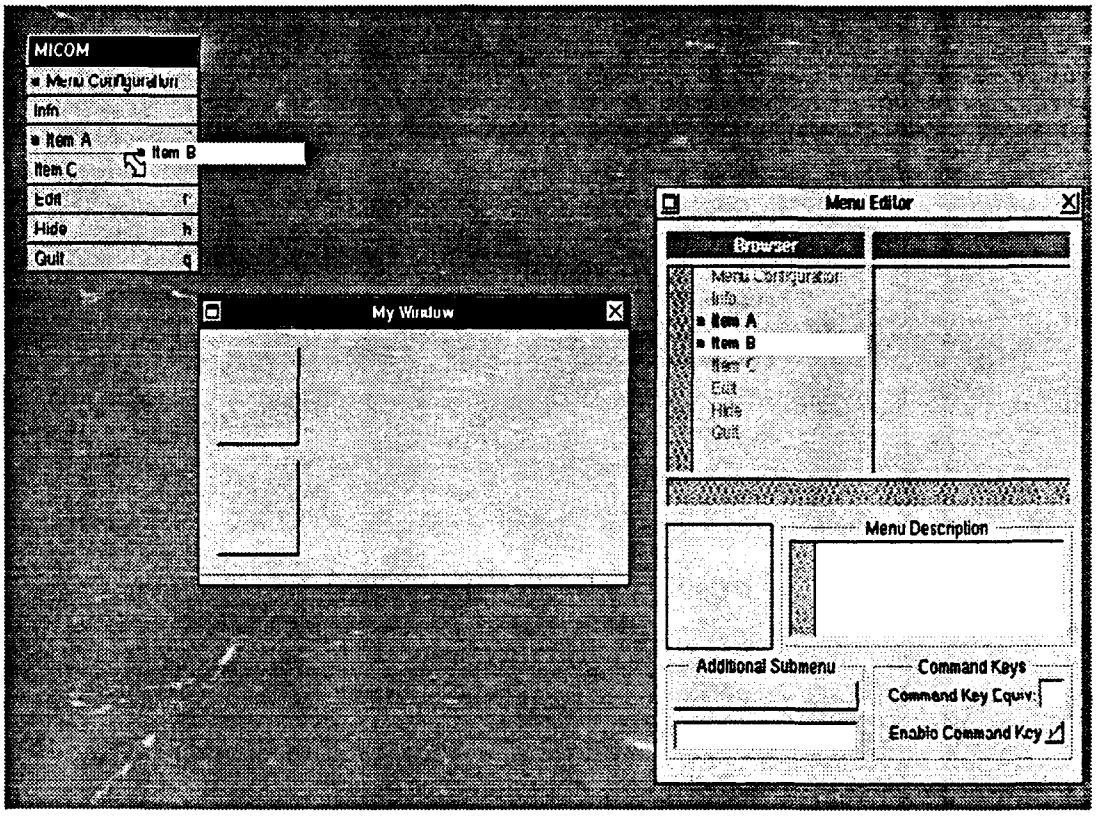


FIGURE 5 Submenu Addition

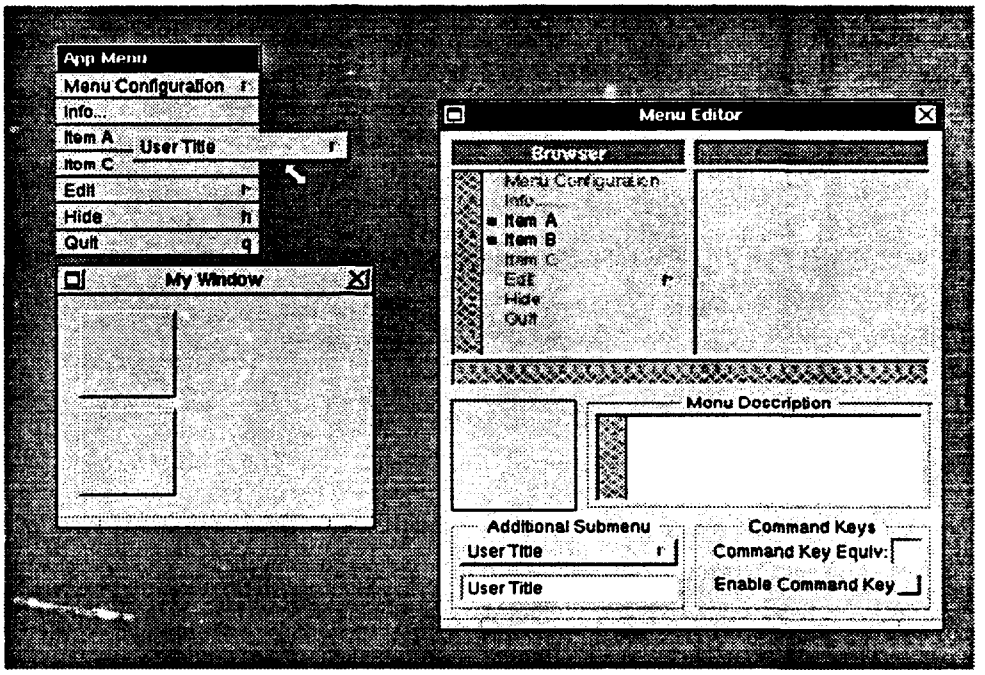


FIGURE 6 Resulting Submenu

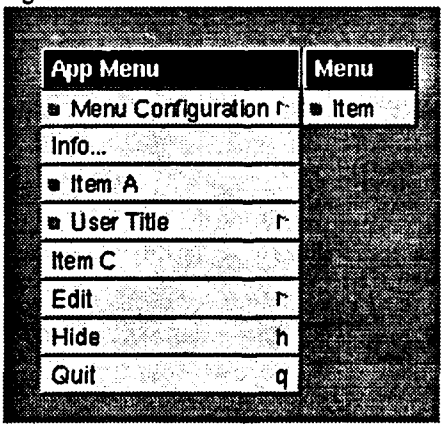
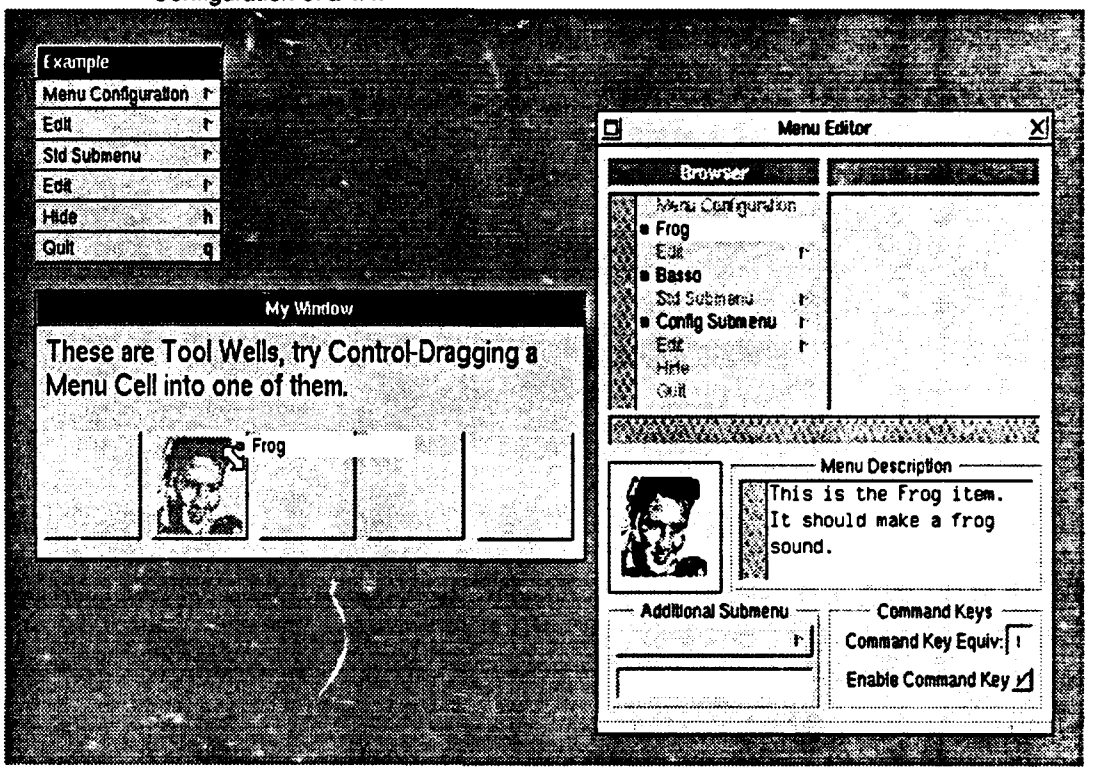


FIGURE 7 Configuration of a TAV



3.0 Architectural Overview

The components and functionalities of the *Command Editor* can be classified into two categories: those necessary for IB and paletting and those necessary at runtime.

3.1 IB and Paletting Components

Palette objects include the **CEController**, **CEMenuCell**, **CESubmenuCell**, **CEMenuConfigCell**, and the **TAV**. Each class conforms to protocols required of the paletting mechanism, associating itself with the correct IB Inspector.

IB Inspectors allow the developer to configure the palette objects. Specifically, the **CEMenuCellInspector** allows specification of **CEMenuCell** fields discussed in Section 2.2. The **CEMenuCellInspector** also allows the specification of "Default" and "Disable" status for **CESubmenuCells**, which are implemented as **CEMenuCells**. Similarly, the **CETAVInspector** allows the specification of the **TAV**'s border.

While the **CEController** is the central control module at runtime, its duties are limited within IB. The **CEController** comes into play when IB saves the developer's application, at which time it composes a translation table to be used at runtime and archives this table to the resulting NIB. We defer the full discussion of this table to Section 5.2.

The **FullMenu** stores the hierarchical structure of the application menu and the information relevant to each menu item. The reader may recall that our original design called for the building of a data structure by the **CEController** that represents this information. Based upon further analysis, we have removed this role of the **CEController**; we extract the information from the **FullMenu** at runtime.

3.2 Runtime Components

This section introduces the major *Command Editor* components present in the application at runtime. To clarify each component's role, we provide a brief description of the component's duties. We omit certain steps for brevity and defer the complete discussion to Section 4.2.

At runtime, the application NIB is unarchived by NEXTSTEP's Application object. The *Command Editor* objects that archived themselves as part of the application NIB are reconstituted: the **CEController** and its translation table, the **FullMenu**, the **Menu Editor**, and the **TAV**s dropped onto the application NIB. The **CEController** is responsible for initializing all *Command Editor* objects and constructing the application menu.

The **CEController** initializes the **FullMenu** by replacing all instances of Menu and Matrix in that structure with **CEMenu** and **CEMenuMatrix**, which contain DND logic. Though the user will never interact with **FullMenu** directly, thus requiring DND in that structure, this replacement facilitates the subsequent task of building the application menu. Building the visible menu involves copying the structure of the **FullMenu**, and the replacement of Menu and Matrix with **CEMenu** and **CEMenuMatrix** saves the **CEController** from performing the same task in the newly constructed menu.

On startup, the application's menu should be replaced by either the user's customized menu or the **BasicMenu**. In the absence of a user profile (*.applicationName*.ceinfo), the **CEController** constructs the **BasicMenu** by replicating a subset of the **FullMenu**. This replication is the filtration by which menu items not marked "Default" are removed from the **BasicMenu**.

In the presence of a user profile, the **CEController** loads the visible menu directly from the file, which is an actual archive of a user modified menu. Each **CEMenuCell** then uses a translation table maintained by the **CEController** to assign its individual target action. This concept will be discussed in detail in Section 5.2.

The **Menu Editor** displays the **FullMenu** and allows the user to customize the application menu. The **Menu Editor** window appears when the user selects "Configure Menus" from the "Menu Configuration" submenu.

The user customizations are stored in the visible menu itself. The user can save this configuration through the "Save Configuration" item. The menu structure simply archives itself to disk, writing out translation table information in the process. This translation table information allows the **CEMenuCells** to reestablish their connection information.

The **TAV**s are unarchived by NEXTSTEP at runtime as views in the application NIB. Subclassed to handle DND directly, the **TAV**s need no initialization from the **CEController**. In the presence of a user profile, however, the **TAV**s require the **CEController** to reestablish connection information. Conversely, when the user saves his customiza-

tions, the **CEController** archives the **TAV** configuration along with the visible menu archive.

4.0 Implementation

4.1 Implementation of Developer Functionalities

This section covers the implementation of classes necessary to provide the developer's suite of functionality described in Section 2.2. We also discuss the implementation of components that assume special roles in IB to provide certain user functionalities. For an introduction to constructing an IB Palette, please refer to the NEXTSTEP on-line document *Building a Custom Palette*.

4.1.1 CommandEditor Class

The **CommandEditor** class, a subclass of IBPalette, is the module responsible for integrating our *Command Editor* classes into IB. The instance data of this class include the palette object types and their iconic representations in the palette window. This class implements a single method, **finishInstantiate**, which initializes all *Command Editor* palette objects for IB.

The *Command Editor* palette provides the following classes to the developer: **CEController**, **CEMenuCells**, **CESubmenuCell**[1], **CEMenuConfigCell**, and **TAV** (Figure 2). The subsequent sections detail the role and implementation of each class relevant to its operation in IB.

4.1.2 CEController

The **CEController** is responsible for creating and archiving the translation tables at the completion of development in IB. Specifically, it traverses (1) the **FullMenu** to compose the **cellTable** and (2) the application's object list to compose the **tavTable**. Both tables are used at runtime to reestablish connection information (Section 5.2). The methods **writeCellTable** and **writeTAVTable** build their respective tables and archive them to the application NIB. Both methods are invoked in the CEController's **write** method, which is automatically executed when the developer selects "Save" in IB. The developer must drag a single instance of the **CEController** from our palette into his application NIB.

4.1.3 CEMenuCell

The **CEMenuCell** is subclassed from **MenuCell** and adds several instance variables to allow the targeted end user functionalities. In IB, the **CEMenuCell** associates its **CEMenuCellInspector** (Figure 8) through the **getInspectorClassName** method. The **CEMenuCell** also implements all methods required of the **CEMenuCellInspector** to set instance data on developer input. Specifically, the **CEMenuCell** can draw its visual in-

1.**CESubmenuCell** is a special case of the **CEMenuCell** class and is not a distinct class in itself (See section 4.1.4)

dicator if "Default" is set to false, update its title when its root state changes, store its tool icon, set its "hasTool" field, and enable/disable its CKE field.

The instance data of the **CEMenuCell** class map to particular functionalities in the following manner.

- States—Each **CEMenuCell** can have multiple states, and each state represents a different title to be displayed once the menu item is selected by the end user. States are stored in a simple list, with one state marked as the root state to be displayed initially. We provide methods that allow the developer to integrate his application logic with this state list. The application is responsible for changing the menu item's state. The **CEMenuCell** automatically updates its title in IB when the developer edits the root state title. Conversely, the root state title in the **CEMenuCellInspector** updates should the developer edit the **CEMenuCell** directly.
- Action—Each **CEMenuCell** performs a specific action when selected by the end user. The mechanism for establishing a target connection is the standard IB method of Control-dragging from the menu item to the target object.
- Textual Description—The developer can provide a description of the menu item by typing directly in the "Menu Description" field in the **CEMenuCellInspector**. This description is available to the user in the **Menu Editor**.
- Icon—The developer can load an image as the icon for a particular **CEMenuCell**. This icon is used to represent the menu item in a **TAV**.
- Has Tool—When true, this boolean field indicates that the **CEMenuCell** has an icon and can be dropped into a **TAV** at runtime.
- Disabled—**CEMenuCells** configured as "Disabled" will disallow the end user's selection of that item at runtime. If "Disabled" is set to true for an item, the menu structure displays that item in "grayed out" fashion.
- Default Status—**CEMenuCells** configured as "Default" are members of the **BasicMenu** and are included in the visible menu loaded at runtime in the absence of a user customization profile. The user can delete **CEMenuCells** from the visible menu; any item the developer considers permanent to the application menu should be integrated as a regular NEXTSTEP **MenuCell** instance.

4.1.4 CESubmenuCell

The **CESubmenuCell** is actually implemented as a **CEMenuCell** whose target is a **CEMenu** (Figure 9). Such **CEMenuCells** answer true to **hasSubmenu** and draw themselves with an additional, right-adjusted submenu arrow. When the developer drops a **CESubmenuCell** onto the application menu in IB, the structure being added consists of the top level **CEMenuCell**, a **CEMenu** that is the target of the toplevel **CEMenuCell**, and another **CEMenuCell** contained in that **CEMenu**. The **CESubmenuCell** associates its inspector, the **CEMenuCellInspector**, with the **getInspectorClassName** method. The instance fields of meaning for a **CESubmenuCell** are "Default" and "Disable." The "Default" status includes the submenu in the **BasicMenu**. The "Disable" status precludes user selection of that **CESubmenuCell**.

FIGURE 8 The CEMenuCellInspector

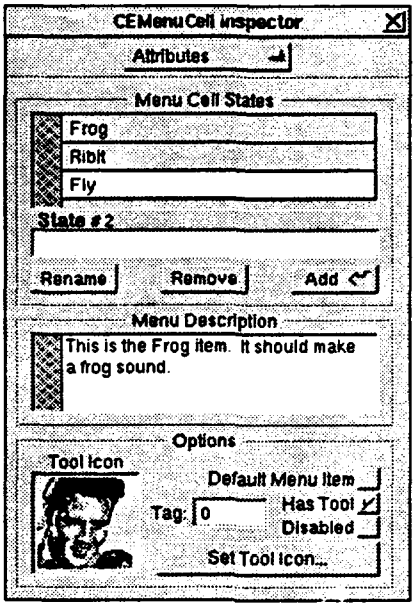
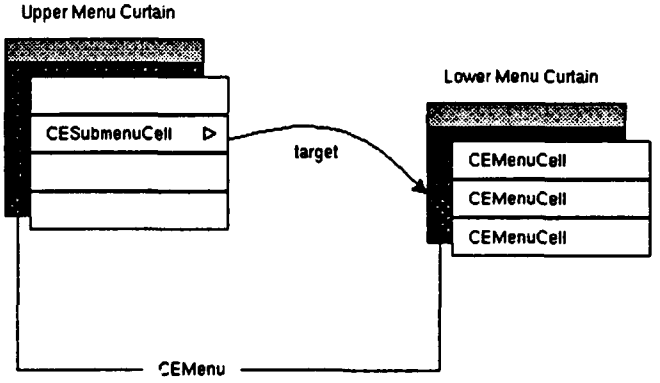


FIGURE 9 The Anatomy of a CESubmenuCell



4.1.5 "Menu Configuration" Submenu

We implement the "Menu Configuration" submenu as a stub object. The developer can drag the item from the *Command Editor* palette onto the application menu. The entity added, however, is simply an instance of **CEMenuConfigCell**. **CEMenuConfigCell** is a subclass of **CEMenuCell** that adds no data or methods (class or instance). In this sense, the **CEMenuConfigCell** class is functionally equivalent to the **CEMenuCell** class. The dropped cell is used as an attach point for the "Menu Configuration" submenu at runtime. In the construction of the **BasicMenu**, the **CEController** searches for a cell of class **CEMenuConfigCell** and attaches the "Menu Configuration" sub-

menu to this cell. The developer should add only one instance of **CEMenuConfigCell** to the application menu, and he will not be able to view the "Menu Configuration" submenu in IB.

4.1.6 TAV

Tool Acceptor Views are instances of the **CEToolView** class. **CEToolView** is a subclass of **CEIconView**, which is itself a subclass of View. The **TAV** class inherits the ability to display an image and respond to input from the View class. The **CEIconView** class, in turn, adds the ability to set the image from a file and display the image in a "ghosted" style. Finally, the **TAV** contains DND logic and can set its instance data given a **CEMenuCell**. Specifically, the **TAV** displays the icon associated with a **CEMenuCell** and, on user selection at runtime, sends a message to the **CEMenuCell**'s target object with the appropriate selector.

The **TAV** associates its inspector, the **CETAVInspector**, with the **getInspectorClassName** method. The developer uses the inspector to specify the border style of the selected **TAV**.

4.2 Implementation of User Functionalities

4.2.1 CEController and the Startup Sequence

The **CEController** is responsible for all initialization and configuration of the menus and **TAV**s on startup. Additionally, the ***CEController*** processes many user customizations committed in the **Menu Editor**.

On startup, the **CEController** executes the following sequence.

1. The **CEController** loads the translation tables (**cellTable** and **tavTable**) containing keys or unique identifiers (UIDs) for the menu and **TAV**s.
2. The **CEController** traverses the hierarchical **FullMenu**, replacing instances of Menu and Matrix with instances of **CEMenu** and **CEMenuMatrix**. The replaced objects are placed on a list for deallocation later.
3. The **CEController** traverses the **FullMenu** once again, loading UIDs into the appropriate **CEMenuCells**. This pass also dictates that the **CEController** create management tables for each MenuCell instance (Section 5.2).
4. The **CEController** then initializes the "Menu Configuration" submenu and loads its four UIDs into the **cellTable** ("Configure Menus," "Save Configuration," "Load Configuration," and "Show/Hide Configurable Cells").
5. The **CEController** builds the visible menu. This process takes one of two paths.
   - **5a.** If the user's profile (*.applicationName*.ceinfo) does not exist in his home directory, we construct and present the **BasicMenu**. To do this, the **CEController** replicates the **FullMenu** with a recursive method called **deepCopy** (Section 5.6). The **CEController** then removes items not marked "Default" from the new copy of the FullMenu with **filterMenuStructure**. The copy is installed as the application menu. We do this filtration once only, and the **BasicMenu** is archived to the application NIB to avoid redundant work in subsequent invocations of the application.

5b. If the user's profile exists, the **CEController** reads the visible menu and TAV configuration from this archive. As part of this load procedure, each **CEMenuCell** restores its connection information using the **cellTable** in the **CEController**. Similarly, the **CEController** uses its **tavTable** to restore connection information for each TAV.

6. The **CEController** attaches the "Menu Configuration" submenu to the newly constructed menu. It then makes this the visible menu with

```
[NXApp setMainMenu: visibleMenu].
```

7. Items on the list composed in Step 2 are deallocated. The necessity for this delayed action is discussed in Section 5.3.

When the user selects "Load Configuration" from the "Menu Configuration" submenu and enters a valid file name in the subsequent dialog box, the **CEController** loads the file, effectively executing step 5b. The existing visible menu is deallocated and the newly loaded menu is set with

```
[NXApp setMainMenu: newMenu].
```

There is no need to attach the "Menu Configuration" submenu; the archived menu structure already contains this submenu. Conversely, the **CEController** archives the menu structure and TAV configuration to disk when the user selects "Save Configuration."

The next section describes the **CEController** responsibilities necessitated by the **Menu Editor**.

4.2.2 Menu Editor

The **Menu Editor** is a window that resides in the **CEController** NIB and is brought forth by the **CEController** when the user selects "Configure Menus" from the "Menu Configuration" submenu. The **Menu Editor** contains several components that allow the user to browse the set of available commands, specify CKEs, and add items to the application menu.

The **CEBrowserMatrix**, which displays the set of available commands, is a subclass of BrowserMatrix. We add DND logic to provide the capability to drag a cell from the browser onto a menu curtain. Each item displayed in the browser is a **CEBrowserCell**. This cell contains a reference to the equivalent cell in the **FullMenu** (Figure 10). When the **CEBrowserMatrix** detects a drag, the selected **CEBrowserCell** is told to write itself to a private pasteboard. The **CEBrowserCell** instructs its **FullMenu** equivalent cell to write its data to this pasteboard, including its UID. On the subsequent drop, the receiving **CEMenu** reads the cell information from the pasteboard. The UID is then used to access the original **FullMenu** member. This member contains a valid connection to its target. This connection information is replicated for the newly read cell (Section 5.2). The same logic allows the user to drag a **CEMenuCell** onto a TAV.

The "Additional Submenu" portion of the **Menu Editor** contains an instance of the **CEAdditionalSubmenuView** class. This subclass of Button implements additional methods for DND. It also allows the specification of its title, and the user can do so by typing in the text field in the "Additional Submenu" portion of the **Menu Editor**. The **CEAdditionalSubmenuView** draws itself with the submenu arrow and contains references to its target **CEMenu** curtain and a single **CEMenuCell** item in that curtain.

When dragged, the **CEAdditionalSubmenuView** writes itself to the Pasteboard, including its referenced instances of **CEMenu** and **CEMenuCell**. On the subsequent drop, the receiving **CEMenuMatrix** curtain reads the entire structure from the Pasteboard and integrates the new submenu containing a single item into the menu structure.

The customization of CKEs requires intervention by the **CEController** (Figure 11). When the user enters a letter in the "Command Equivalent Key" field of the **Menu Editor**, the **CEController** follows a reference contained in the selected **CEBrowserCell**. This reference points to a member of the **FullMenu**—a **CEMenuCell**. The **CEController** then uses the UID of this **CEMenuCell** to collect members of the visible menu that are equivalent to this **CEMenuCell**. The **CEController** configures these instances to activate on the new CKE. If no such instances are found, the **CEController** ignores the new CKE. The entry in the **FullMenu** is not modified; the next selection of the same command will reflect the CKE originally set by the developer. The customization of the "Enable/Disable CKE" field is analogous.

FIGURE 10 The Association Between CEBrowserCells and CEMenuCells

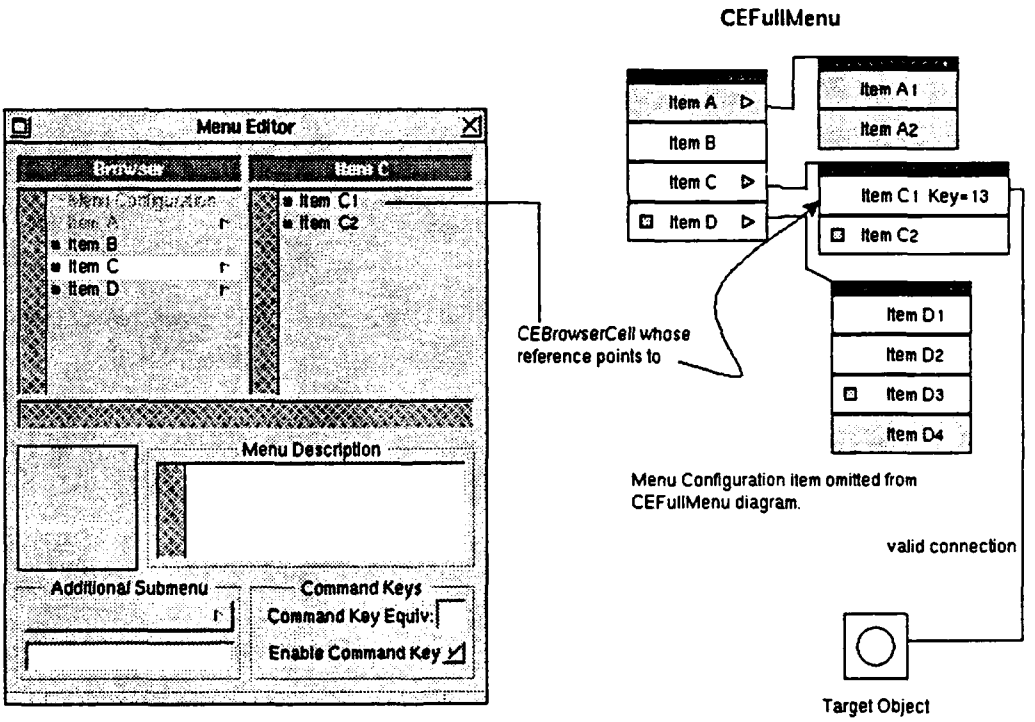
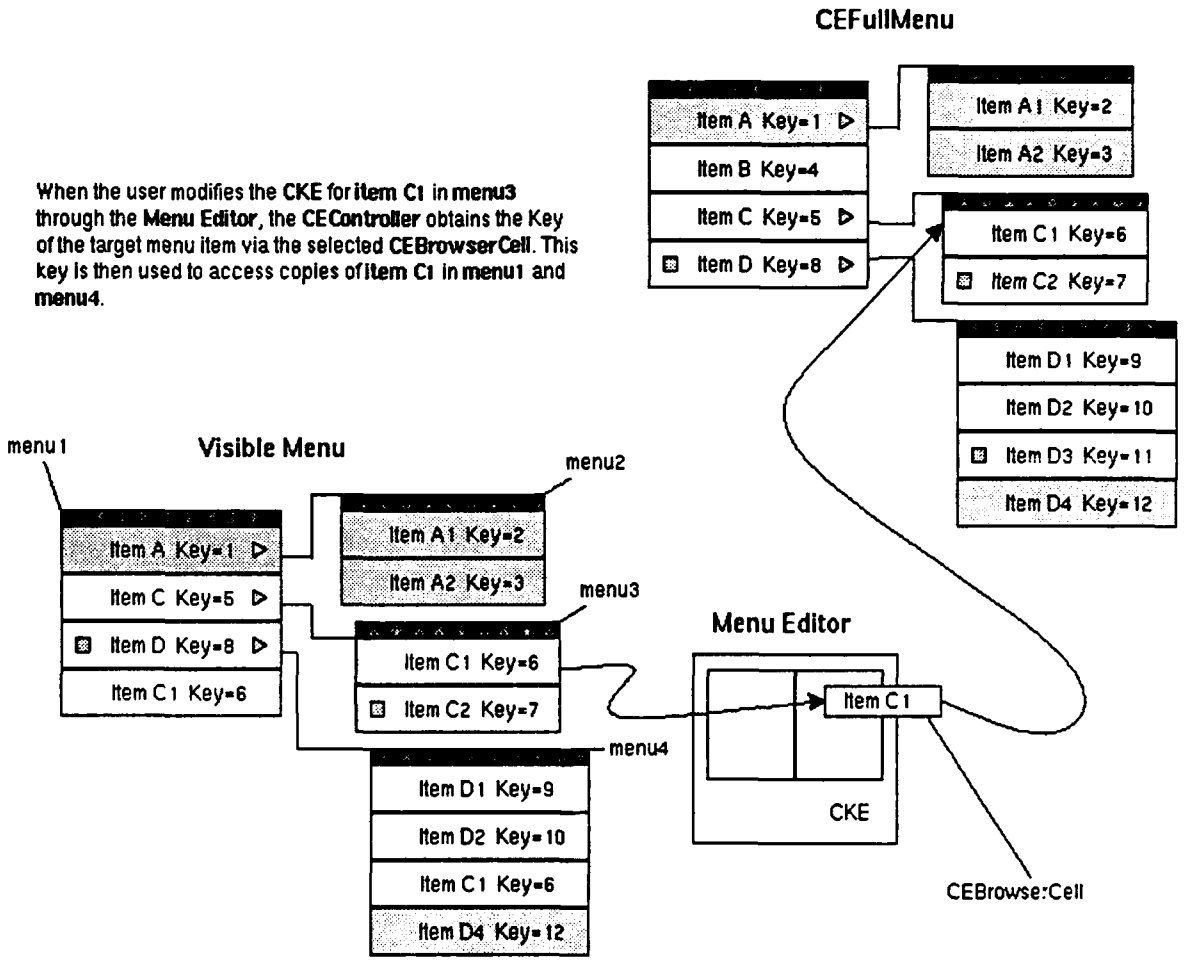


FIGURE 11 References for CKE Updates



4.2.3 DND in Menus

The provision of DND logic in the application menu entails replacement of instances of Menu and Matrix with instances of **CEMenu** and **CEMenuMatrix**. On a Control-drag, the **CEMenuMatrix** will detect the drag and map the mouse coordinates to one of its members—a **CEMenuCell**. This cell implements **read** and **write** methods that allow it to initialize itself from the pasteboard and format itself to the pasteboard, respectively. The **read** restores the **CEMenuCell**'s target action using the **cellTable**.

The **CEMenu**, then, is used to manage instances of MenuCell. The MenuCell class cannot read or write without losing its connection information (Section 5.2). The **CEMenu** maintains a table of [UID, MenuCell reference] pairs that facilitates the restoration of each MenuCell's target action.

4.2.4 DND In TAVs

We implement TAVs as a subclass of View. On a simple **mouseDown**, a **TAV** sends its target object the appropriate selector. A **TAV** stores its target and selector information in a **CEMenuCell** instance datum. On a Control-drag, a **TAV** writes its **CEMenuCell** to the Pasteboard and invalidates its **CEMenuCell** instance datum. The visual indicator is the "ghosted" icon on the **TAV** itself. On a drop, the receiving **TAV** reads the **CEMenuCell** from the Pasteboard and assigns its **CEMenuCell** reference to this newly allocated cell.

5.0 Challenges and Resolution

The challenges we faced in completing the *Command Editor* stem from two aspects of the NEXTSTEP environment. The first aspect is the process by which NEXTSTEP archives an application package (executable and NIBs) and the subsequent unarchival (reconstitution) of the application's objects at runtime. The second aspect is NEXT's private API that hides some functionalities of IB from the developer.

5.1 MenuTemplate

The NEXTSTEP implementation of menus is hidden from the developer in IB. IB deploys a "MenuTemplate" hidden in an API private to NEXT to implement the menu structure. Consequently, our original design could not be implemented. The reader may recall that DND was to be achieved by replacing the Matrix instances of the menu with instances of our **CEMenuMatrix** in IB. While we indeed perform this replacement, this action is done to the **FullMenu** on startup. As a result, we now construct the visible menu from the **FullMenu**, filtering out non-"Default" members of the hierarchy.

5.2 Restoring Connection Information

The standard NEXTSTEP API provides the **write** method for the archival of objects. This **write** method formats data to a stream and can be used to archive an object to disk. Data written include instance variables and methods. If the object contains references to other objects, however, these references are not written and read correctly unless the referenced objects are likewise written to the same stream. The **write** method actually writes out a preset value for all references to external objects. The **read** sets the reference to NULL when it encounters the preset value, effectively destroying the connection information.

In a typical application developed in IB, connection information is archived in the NIB that encapsulate the objects involved in that connection. This archival method is private to NEXT and presumably does much more than the **write** method. At runtime, NEXTSTEP unarchives the NIBs and establishes the connection information correctly.

Our challenge stems from the need to retain the connection information in an object despite using the **write** method. Specifically, we require the **write** mechanism to enable DND; an object with external references (a **CEMenuCell** with a target object) is written to (and read from) a private pasteboard for each DND session. The same mecha-

nism saves the end user's customizations of the application menu when the customized menu recursively writes itself to disk.

Each **CEMenuCell** has a target object to which it will send a selector when activated by the end user. The developer specifies this connection in IB by Control-dragging an IB wire from the **CEMenuCell** to its target object. At the completion of development in IB, the **FullMenu** structure archives to the application NIB. The structure of this **FullMenu** is fixed by the developer and remains constant throughout invocations of the application. At runtime, NEXTSTEP reconstitutes the **FullMenu** with valid connections and targets. Hence, all the **CEMenuCells**, their target objects, and connection information exist and are valid at runtime.

If we ask a **CEMenuCell** to **write** itself, we essentially lose the reference to its target object. To facilitate DND and profiling of the end user's customization, we implement a translation table that pairs each **CEMenuCell** in the **FullMenu** with a unique key (UID). At the completion of development in IB, the **CEController** traverses the application's object list and inserts [UID, **CEMenuCell** reference] pairs for every **CEMenuCell** instance into a translation table. The **CEController** then archives this table in the application NIB. At runtime, the table is instantiated and its object references point to items in the instantiated **FullMenu**. Each **CEMenuCell** instance in the **FullMenu** uses the table to assign its UID to an internal variable for future use.

When a drag occurs at runtime, the dragged **CEMenuCell** writes itself to the drag pasteboard. On the drop, the **CEMenuCell** is read from this pasteboard, complete with a **UID**. The **CEController** uses the UID to look up the object reference in the translation table. This object reference yields the original **CEMenuCell** in the **FullMenu**. Given this original **CEMenuCell**, which contains the valid reference to its target object, the target connection can be restored.

Similarly, when the end user saves his profile, the customized menu recursively writes itself to disk. The unique key is written for each **CEMenuCell**. On a subsequent run of the application, the **CEController** loads this archive. Since this a complete archive of an object, the target actions do not need restoration.

We treat **TAVs** in a similar manner. At the completion of development in IB, the **CEController** obtains the list of all objects and traverses it looking for objects of class **TAV**. The **CEController** composes a second table to store **[tavUID, CEMenuCell** copy] tuples.

Instances of MenuCell further complicate the connection restoration. Recall that one of our original design objectives was to retain the full functionality of NEXTSTEP, particularly IB. We allow the developer to use the default IB MenuCell to compose the application menu. The end user can drag a submenu from the **Menu Editor** (onto the application menu) that contains MenuCell instances. The connection for the MenuCell is lost when the submenu recursively writes itself to the pasteboard. A key is used in this case to restore the connection, much like for **CEMenuCells**. The difficulty is now associating the key with the MenuCell to be written. The MenuCell is a NEXTSTEP "appkit" class. The **UID**, then, is stored in the **CEMenu** which contains that MenuCell. Each **CEMenu** in the application menu structure must manage a set of MenuCells (and SubmenuCells). For each MenuCell, the **CEMenu** will maintain the unique key that refers to the original MenuCell in the **FullMenu**.

Submenus are a special case for archival and restoration. When a submenu item writes itself, the entire recursive structure is written to the stream. This includes the submenu curtain as well as the menu items that curtain contains. Though the targets of the menu items are lost via the standard **write**, the target of the submenu item is the submenu curtain, which is written to the archive along with the upper level submenu item. No unique key is needed to reestablish the connection. When a submenu item is read, the target object is set to the submenu curtain (**CEMenu**) automatically. As curtain items are read back, we use unique keys to restore their individual connections.

5.3 Race Conditions

A peculiar race condition occurs in the process by which NEXTSTEP instantiates an application's NIB at *runtime*. Typically, an object receives an **awakeFromNib** message once it is instantiated by NEXTSTEP. This message signifies that all objects in that NIB are instantiated and valid. The developer can implement this method to perform any initialization or synchronization desired. In a NIB that consists of many objects, the order in which objects receive the **awakeFromNib** message is not known. Therefore, any dependency in an object's **awakeFromNib** method on another object having received the **awakeFromNib** message is prone to error.

This aspect of the instantiation process gives rise to a race condition that the **CEController** must address. When the **CEController** receives its **awakeFromNib**, it proceeds to replace recursively the **FullMenu**'s **Menu** and **Matrix** instances with instances of **CEMenu** and **CEMenuMatrix**. Replacement entails freeing the replaced Menu and Matrix instances. *While the* Menu and Matrix instances *certainly exist*, many of them have yet to receive their **awakeFromNib** message. If we free these instances before the **awakeFromNib**, the unarchival process will eventually send messages to non-existent objects, causing the application to crash.

The work around for this race condition entails placement of the objects to be freed on a list and, finally, freeing of this list once the **awakeFromNib** is sent to each object. We accomplish the delayed freeing of objects by placing the action on the application's event queue, with the guarantee that this event queue entry will be processed only after the current event is finished. The current event is the unarchival of the NIB. Hence, we free the replaced **Menu** and **Matrix** instances only after the unarchival process is completely finished.

5.4 Submenu Addition

The addition of submenus to the application menu by the end user at runtime dictates that we implement a method by which a DND submenu structure is added to a menu curtain. Recall that the end user can add submenus in one of two ways: by dragging the "Additional Submenu" item from the **Menu Editor** onto the application menu or by dragging a menu item that represents a submenu from the **CEBrowserMatrix** in the **Menu Editor**.

When the "Additional Submenu" item is dragged from the **Menu Editor**, a submenu structure consisting of one item is written to the drag pasteboard. The structure written includes the upper item, its submenu curtain, and the singular lower member item. The submenu curtain is the target object of the upper menu item. On the subsequent drop, the drop-curtain's **CEMenuMatrix** allocates a new **CEMenuCell** which reads itself

from the drag pasteboard. This read process restores the connection information; the submenu curtain is once again the target object of the upper menu item. The **CEMenuMatrix** integrates the new submenu item into its list of items.

When a menu item representing a submenu is dragged from the browser area of the **Menu Editor**, the process by which the new submenu gets incorporated into the existing menu is identical to that of the previous case. In this scenario, however, the items in the submenu curtains have connections that must be restored. Each **CEMenuCell** can restore its own connection information with data written on the **write** with its **read** method. Hence, the process by which the new submenu item is created and initialized from the pasteboard also restores the member items' connections.

5.5 Populating the Matrix

Populating a matrix poses a special problem for our development effort. To add a cell to a Matrix, The NEXTSTEP interface to its "appkit" Matrix class dictates that we first ask the Matrix to create a new cell. Second, we assign instance variables of that cell explicitly. The problems with this mechanism are (1) the cell is always added at the end of the list (bottom of menu curtain) and (2) we must copy a record structure to assign cell instance data. We need the capability to insert a pre-allocated cell into the Matrix at any position.

To this end, we have implemented a method which inserts a cell in the Matrix without copying the structural data explicitly. The method accomplishes this by pointer manipulation. First, the Matrix is asked to allocate a new cell. We then obtain the pointer to this cell, reset the pointer to our cell, and free the newly allocated cell. We have just avoided the explicit copy, but our cell is not at the desired location. Next, we traverse the Matrix's list of pointers and insert our cell in the correct location by shifting the appropriate pointers.

5.6 Deep Copy

The duplication of an existing recursive data structure requires the recreation of each member node. While an object that contains references to additional objects can certainly duplicate itself via the **copy** method, the new copy contains references to the same instances pointed to by the original object, as the following NEXTSTEP on-line documentation clearly shows.

> **copy**
>
> **- copy**
>
> Returns a new instance that's an exact copy of the receiver. This method creates only one new object. If the receiver has instance variables that point to other objects, the instance variables in the copy will point to the same objects. The values of the instance variables are copied, but the objects they point to are not.

Our duplication of a subset of the **FullMenu** to provide the application menu at runtime dictates that we duplicate each member menu cell. If we were to employ the standard **copy** mechanism, the new menu will contain references to the original menu cells in the **FullMenu**. When the end user customizes the menu, modifications made to the newly created menu will alter the configuration of the **FullMenu**.The original hierar-

chy and instance data provided by the developer will be lost. We cannot allow this to happen. This is the motivation for the **deepCopy** method, which recursively recreates each node in a given menu structure and replicates all referenced objects.

6.0 Compromises to Functionalities

6.1 States with Individual Targets

Our original design allowed each state of a **CEMenuCell** to have a distinct target action. We assumed that a mechanism exists in IB that would allow the developer to establish the connection from the individual state to its target object. This is typically done through the Control-drag mechanism that displays an IB wire. Unfortunately, the IB connection establishment mechanism is private to IB. All states will represent the same action.

6.2 System Menus

The "Windows" and "Services" menus are maintained by the system. As such, their items are determined dynamically over the course of the system's execution. We cannot add DND to these system menus; doing so will potentially cause a plethora of errors that affect components external to the application.

Appendix A Class Descriptions

This section lists the classes of the Command Editor: those used by the developer as and those necessary to construct the Command Editor itself. Each entry contains the class and category name, its superclass if applicable, the file that contains the implementation, and a brief description of its purpose and use.

className(*category*): ***superClass*** in fileName

description

Fields that are inapplicable for an entry are denoted with a "-". Categories group methods by common purpose and separate logically distinct sets of methods. A category entry begins with the class name followed by the parenthesized category name. The entries appear in alphabetical order.

Application*(CEAdditions)*: - in Applications_CEAdditions.[mh]

This category adds one method to the standard Application class. The **CEController** uses this method, **setWindowsList**, to obtain the list of windows in the application and access the application menu at runtime.

CEAdditionalSubmenuView: ***Button*** in CEAdditionalSubmenuView.[mh]

This class implements the draggable submenu button in the **Menu Editor**'s "Additional Submenu" portion. The end-user can set the title of this submenu and drag the button into the menu structure. On the drop, the new submenu contains a single configurable item.

CEBrowserCell: *NXBrowserCell* in CEBrowserCell.[mh]

CEBrowserCells are used in the **Menu Editor** to allow the user to browse the set of available commands. Each **CEBrowserCell** contains a proxy or reference to its representative in the **FullMenu** and displays the title of its proxy's root state.

CEBrowserMatrix: ***CEDragMatrix*** in CEBrowserMatrix.[mh]

The **CEBrowserMatrix** inherits DND logic, allowing the end-user to drag a **CEBrowserCell** from the **Menu Editor** into the application menu. Most of the implementation of the DND capability is in the abstract superclass **CEDragMatrix**. The **CEBrowserMatrix** supplies the verification methods to tailor the drag sessions. Specifically, the end-user must hold down the Control key to start a drag. In addition, the **CEBrowserMatrix** does not accept drops.

CEController: ***Object*** in CEController.[mh]

This section of the **CEController** contains the methods necessary to execute the sequence detailed in Section 4.2.1. These methods handle the archival and unarchival of the three menus (**FullMenu**, **BasicMenu**, and **CustomizedMenu**) and build, maintain, and manipulate the UID translation tables (**cellTable** and **tavTable**).

CEController*(IB)*: - in CEController_IB.m

This category of **CEController** contains methods to write the UID tables (**cellTable** and **tavTable**) to the application NIB at the completion of development in IB.

CEController*(UserDefaults)*: - in CEController_UserDefaults.m

This category of **CEController** contains methods to load and save the end-user's configurations. The **CEController** loads the **CustomizedMenu** and configures the TAVs using methods in this category.

CEController*(UserInteraction)*: - in CEController_UserInteractions.m

Methods in this category implement the selections in the "Menu Configuration" submenu. The **CEController** uses methods in this category to bring forth the **Menu Editor** and show or hide configurable cells. This category additionally handles all end-user customizations done through the **Menu Editor**: specification and enabling of CKEs, traversal of the **FullMenu** through the browser, selection and dragging of **CEBrowserCells**, and addition of a submenu.

CEDragMatrixView: ***Matrix*** in CEDragMatrixView.[mh]

This is the abstract superclass for **CEBrowserMatrix** and **CEMenuMatrix**, both of which require different DND capabilities. This class implements the DND methods and leaves stub methods that a subclass must override to tailor the DND capabilities. To facilitate the drag, the **CEDragMatrixView** accepts the mouseDown event, maps the mouse coordinates to the draggable item, obtains the drag image, obtains the drag data, and writes that drag data to a private pasteboard. To facilitate the drop, the **CEDragMatrixView** updates the cursor on **draggingEntered** and **draggingUpdated** and reads data from the private pasteboard on the actual drop.

CEIconView: ***View*** in CEIconView.[mh]

This is the abstract superclass of **CEToolView** (which implements TAVs). **CEIconView** contains methods to display an image centered in a view with options for background style and ghosting.

CEMenu: ***Menu*** in CEMenu.[mh]

The **CEMenu** manages its **MenuCells** and their targets to allow the restoration of connection information. In addition, the **CEMenu**, at the request of the **CEController**, does the actual replacement of Matrix with **CEMenuMatrix** for itself and its submenus. The **CEMenu** also implements methods to add and remove submenus, update CKEs, and update enable/disable display status of its items.

CEMenuCell: ***MenuCell*** in CEMenuCell.[mh]

An instance of **CEMenuCell** is the atomic datum that represents a command in the menu structures. The **CEMenuCell** inherits basic menu item functionality from MenuCell. Additional instance data record its list of states, icon, description, "hasTool" status, "Default" status, CKE, and UID. **CEMenuCell** also contains methods to access, manipulate, and update its instance data. The **CEMenuCell** can display itself with the square notch to indicate either default (IB) or configurable (runtime).

CEMenuCell*(IB)*: - in CEMenuCell_IB.m

This category contains a single method **getInspectorClassName** that associates the **CEMenuCellInspector** with the **CEMenuCell** class. This allows the developer to configure the instance data of a **CEMenuCell** in IB.

CEMenuConfigCell: ***CEMenuCell*** in CEMenuConfigCell.[mh]

This is a stub class used for the attachment of the "Menu Configuration" submenu (Section 4.1.5).

CEMenuMatrix: ***CEDragMatrix*** in CEMenuMatrix.[mh]

This class allows the end-user to drag items from and drop items onto the application menu. **CEMenuMatrix** overrides its superclass method **validateBeginDrag** to allow the end-user to drag **CEMenuCells** from the application menu. Similarly, it overrides its superclass method **validate:andPerformDrop** to allow the end-user to drop **CEMenuCells** into the application menu.

CEState: ***ButtonCell*** in CEState.[mh]

An instance of **CEState** maintains a title, a numeric identification tag, and a boolean field that reflects whether or not this instance is the root state. Such instances are grouped into a list and stored in an instance datum of the **CEMenuCell** class. The developer can access a **CEState** by its numeric identification tag (index) and update the menu item's displayed title with the **CEState**'s title.

CEToolView: ***IconView*** in CEToolView.[mh]

An instance of **CEToolView** contains an instance of **CEMenuCell** that s' es the **CEToolView's** configuration. When the user selects the **CEToolView** instance (**TAV**), the **TAV** accesses its instance of **CEMenuCell** to obtain the target object and the selector to send to that target object. This action effectively executes the configured command. In addition, **CEToolView** contains methods to implement the DND configuration capabilities.

CEToolView*(IB)*: - in CEToolView_IB.m

This category contains a single method **getInspectorClassName** that associates the **CEToolViewInspector** with the **CEToolView** class. This allows the developer to specify the border style of a **CEToolView** in IB.

Cell*(CEAdditions)*: - in Cell_CEAdditions.[mh]

This category adds methods to the standard NEXTSTEP class Cell to obtain and set a single character value. These methods facilitate the specification and updating of CKEs.

Matrix*(CEAdditions)*: - in Matrix_C' ,dditions.[mh]

Methods in this category implement the work around described in Section 5.5.

NXImage*(CEAdditions)*: - in NXImage_CEAdditions.[mh]

This category adds the method **initFromRect** to allow the initialization of a NXImage from a region of a view. We use this capability to provide the dragging image when a **CEBrowserCell** is dragged.

Object*(CEAdditions)*: - in Object_CEAdditions.[mh]

This category contains a single method **deepCopy** to replicate an object as well as the objects it references (Section 5.6).

Appendix B Related Documents

A Command Editor Tool for NEXTSTEP and X-Windows Systems (SBIR Phase 1 Proposal) for DARPA/OASB/SBIR, Submitted on July 1, 1992.

A Command Editor Tool for NEXTSTEP and X, Quarterly Status Report under Contract # DAAH01-93-C-R013, Submitted on April 26, 1993.

A Command Editor Tool for X and Motif, Design Document under Contract # DAAH01-93-C-R013, July 1, 1993.

The Command Editor: A Manual for Users and Developers, under Contract # DAAH01-93-C-R013, July 1, 1993.

End-user Customizable Menus in the X Windowing Environment (SBIR Phase II Proposal) for Contract # DAAH01-93-C-R013, July 1, 1993.

Appendix C Sources

Building a Custom Palette	/NextLibrary/Documentation/NextDev/DevTools/18_CustomPalette
Interface Builder	/NextDeveloper/Demos/HeaderViewer.app
IB (protocol)	/NextLibrary/Documentation/GeneralRef/08_InterfaceBuilder/Protocols/IB.rtf
IBConnectors	/NextLibrary/Documentation/GeneralRef/08_InterfaceBuilder/Protocols/IBConnectors.rtf
IBDocuments	/NextLibrary/Documentation/GeneralRef/08_InterfaceBuilder/Protocols/IBDocumnets.rtf
IBDocumentControllers	/NextLibrary/Documentation/GeneralRef/08_InterfaceBuilder/Protocols/IBDocuments.rtf
IBEditors	/NextLibrary/Documentation/GeneralRef/08_InterfaceBuilder/Protocols/IBEditors.rtf
IBInspectors	/NextLibrary/Documentation/GeneralRef/08_InterfaceBuilder/Protocols/IBInspectors.rtf
IBObject	/NextLibrary/Documentation/GeneralRef/08_InterfaceBuilder/Protocols/IBObject.rtf
IBSelectionOwners	/NextLibrary/Documentation/GeneralRef/08_InterfaceBuilder/Protocols/IBSelectionOwners.rtf
Matrix	/NextDeveloper/Headers/AppKit/Matrix.h
Menu	/NextDeveloper/Headers/AppKit/Menu.h
MenuCell	/NextDeveloper/Headers/AppKit/MenuCell.h
NXBrowser	/NextLibrary/Documentation/GeneralRef/02_AppKit/Classes/NXBrowser.rtf
NXBrowserCell	/NextLibrary/Documentation/GeneralRef/02_AppKit/Classes/NXBrowserCell.rtf
NXDraggingDestination	/NextLibrary/Documentation/GeneralRef/02_AppKit/Protocols/NXDraggingDestination.rtf
NXDraggingInfo	/NextLibrary/Documentation/GeneralRef/02_AppKit/Protocols/NXDraggingInfo.rtf
NXDraggingSource	/NextLibrary/Documentation/GeneralRef/02_AppKit/Protocols/NXDraggingSource.rtf