

Information Interactions:
User Interface Objects
for
CODER, INCARD, and MARIAN
v. 2.5

Robert France

24 August 1992

Table of Contents

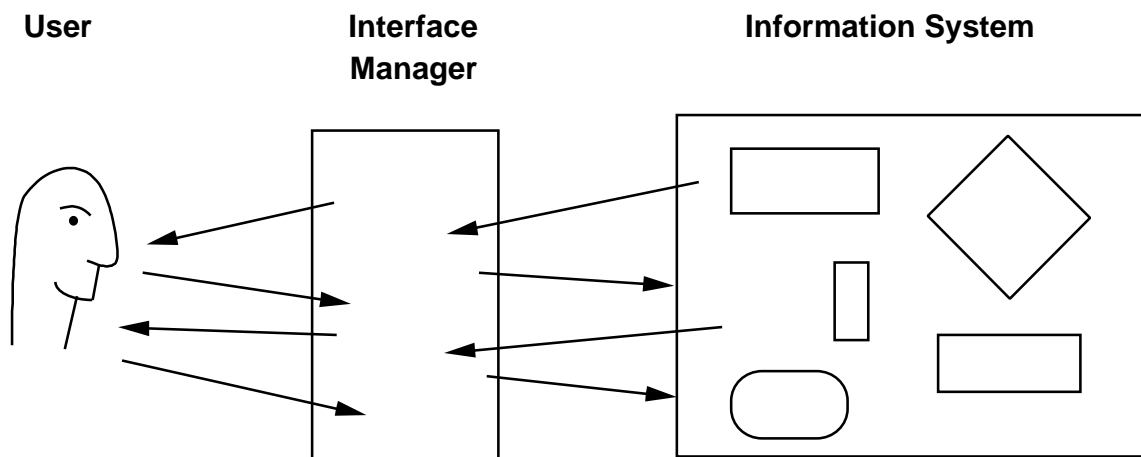
Introduction	2
About This Document	5
General Notes	8
Implementation Notes	13
Interface Object	14
Base Object	18
CODER	20
MARIAN	21
Modal Objects	23
Parent Modal Object	25
Information Display	26
Yes/No Question	27
Information Question	28
User Identification	
CODER Identification	29
MARIAN Login	30
Non-Modal Objects	31
Parent Non-Modal Object	34
Text Object Display	36
Text Object Creation	38
Query Creation	40
Journal Article Form Query Creation	42
Bibliographic Form Query Creation	44
Simple Bibliographic Form Query Creation	47
Text Collection Display	49
Book Display	51
Choice List Display	55
Text-valued Hypernode (Text Node) Display	57
Text Node Collection Display	59
MeSH Node Collection Display	61
Retrieval Set Display	64

Introduction

Any information system needs a user interface: a program or program module that eases the communication between the system's users and the underlying search and storage software. This document describes (part of) the specifications for the user interface to a family of information systems current at Virginia Tech: the experimental platform CODER, a specialized version of CODER dealing with medical information called INCARD for INformation about CARDiology, and a library catalog system named MARIAN.

All of these systems share certain characteristics. They are all text-based, with advanced methods for dealing with English language documents. They each store several different types of information, including different types of documents, highly formatted data, and semantic networks. They all make use of the imagery of hypermedia to present information as objects connected by links. And they are all implemented as networked sets of communicating processes. One such process is the user interface manager.

The user interface manager used by these systems takes the form of an autonomous process that runs in parallel with the information system. The sole responsibility of the interface manager is to present system actions to the user and user actions to the system.



The interaction between user and system can be thought of as a conversation, and like any human-to-human conversation it contains many threads. The user may initiate a search on the system's underlying information resources, remember a second topic that s/he wants searched before the results of the first are back; ask for help; refer back to h/ir first query, and so forth. The system may prompt for clarification on a misspelled word; present partial results for the first search; ask the user to narrow h/ir query to a more precise description; present help; present warning messages; add further discoveries to its partial list, and so on. These messages are interleaved in arbitrary ways: we say that this sort of interface facilitates a *mixed-initiate* system.

We organize this apparently chaotic interaction in two ways: by invoking the discipline of object-oriented programming, and by delving deeper into our understanding of conversations.

Like a human conversation, the interaction between a user and an information system can be seen as a sequence of acts, each of which occurs within the context of the preceding conversation. Thus each act by the user either opens a new line of inquiry into the system's information resources – including perhaps its information on its own states and operation – or furthers an existing line of inquiry; and each act by the machine opens up new possibilities for user acts. Except for global considerations like screen management, it is best to ignore the conversation as a whole in favor of the concurrent threads that make it up. Each thread can then be considered separately, as a linear series of conversational actions.

Human actions in conversations – what philosophers sometimes refer to as *speech acts* – can be divided into three categories: statements, questions and commands. We generally consider it impolite for machines to give humans commands, and it is relatively unusual for a user to offer unsolicited information to a machine, so the conversations carried by our user interfaces can be thought of as sequences of system statements and questions interspersed with user questions, commands, and occasional statements.

The medium for this conversation is a continuous flow of objects created by the user interface in response to user and system acts. These *interaction objects*, as we shall call them, are objects in the object-oriented programming sense. This means two things: that they are grouped into classes within an inheritance hierarchy, and that each is completely defined by a certain amount of data and by the *methods* that it presents to deal with the data. The classes we define and the hierarchy into which they fit are the subject of the body of this document, and should be easy going for any object-oriented programmer. The unique aspect of interaction objects is the way that we treat each class's methods.

The functionality associated with an interaction object has two parts: the computational functions that the object presents to the information system and the user interface program, which we shall call the *methods* for an object of the class, and the functions that it presents to the user, which we shall call the possible *user actions* for the class. Each interaction object carries a certain amount of information in itself – a message from the system; a document or set of documents; a form for the user's next query – and it carries a set of possible things the system can tell it to do – clear its message field; add new text to the end of a document; disappear – but it also carries a spectrum of user acts enabled by the object.

This makes the object a carrier of context as well as information. Just as the context of a human conversation constrains the possible responses to any act of one of the participants, an information object constrains the "next moves" that the user can make within a conversation thread. These moves typically include continuing the thread – getting further results to an ongoing search, for instance – starting a new thread – taking one of the search results as the starting point for a different search – and referring to the thread itself – getting help or asking the system the status of the search. What is denied is the possibility of applying arbitrary actions in arbitrary contexts.

This approach must certainly be supplemented with a certain amount of global conversation management. It must be possible, for instance, for the user to start a totally new thread, unrelated to any that has gone before. This is the responsibility of the Base interaction object class. It must also be possible for the user to return to objects from earlier in the conversation, using them as branch points to start new threads. This sort of navigation is not handled by any of the interaction object classes described in this document, although if the objects are implemented as windows, much of it can be left to the window management system. Similarly, it must be possible for the system to invoke a method of an object that was created earlier in the conversation, even though many objects of its class may be active at the time. We have made the convention that each object within a class is uniquely determined by a string: its message for modal or title for non-modal objects. Thus a class manager – or the user interface process itself – can always determine the object to which a system method should be referred. How that is done, however, is also left to the interface implementor.

The purpose of this document is simply to define the classes of interaction object needed by the family of information systems that include CODER and MARIAN. It defines two important aspects of the system: what moves, at any point in a user-system conversation, are legal for the user to make, and what moves are legal for the system to make. Thus it is both a design document and an implementation document. It serves the system designer by providing a concrete framework within which the capabilities of a system can be defined or extended, and it serves the implementor by specifying the low-level objects from which the user interface is built. The remainder of the interface is left to the ingenuity of the implementor.

About This Document

1. This document describes the classes of objects that mediate interactions between users and the CODER family of systems. Most of the categories of objects and interactions are common throughout members of the CODER family. Some few, though, have been specialized to deal with information objects from the medical domain explored by the INCARD incarnation of CODER or bibliographic objects from the MARIAN library system.

Each class of interface objects is described two ways: as an abstract interaction and as a concrete object presented to the user. Since current interface development in CODER is concentrating on the X Window System, the concrete objects are all interpreted here as windows. It is to be stressed, however, that the objects are also intended to be used in non-windowing systems. Each class description is accompanied by a drawing of how a member of that class might look in X. These drawings are preliminary, and meant to give only rather broad hints on the final look and feel of the windows. They do, however, include all the functionality required for each class. Thus, while any real implementation of the classes is expected to have at least a slightly different look and feel, all implementations should include all the described functionality.

Each class of abstract interaction is represented by a functional profile of the class. Both the methods of the class presented to the body of CODER (the "call-ins") and the methods of CODER invoked by the interaction object (the "call-backs") are detailed. This latter group is particularly important for the abstract definition, as it defines the categories of user actions that effect the system outside of the interface. As of this version (2.4), abstract descriptions of the user actions are somewhat lacking, and must be inferred from the X menus in many classes. Please note, though, that the actions do not need to be implemented as menus unless it is idiomatic, as it is in X, to use menus for the local functionality of individual objects. Buttons, pop-ups, function keys, commands lines would all do as well.

2. After some short introductory matter, the document is separated into two sections, one on modal and the other on non-modal windows. Each section begins with an inheritance diagram for the interaction objects it covers. The remainder of the section is organized as an in-order traversal of the class hierarchy. Classes with multiple parents are visited after the last of their parents has been defined, so that the reader will be familiar with all aspects of the class's heritage.
3. Each class description is laid out as follows. At the top of the first page is the name of the class. Then follows a declaration in C for the object-creating function or constructor (className::className() in C++ terms). An example use of the function is followed by a drawing of the window that that example would produce in an X implementation. A section of notes follows; these are not intended to substitute for a close inspection of the drawing or for the use of the reader's common sense, but to clarify points that may not be obvious. Further sections, as appropriate, detail the possible user actions, the callbacks invoked, any additional methods that the class supports, and class siblings and children.

4. Not every class in this description is intended to be used in an actual CODER system: some are described for convenience of definition. For instance, neither the parent modal nor the parent non-modal class is designed to be implemented, but both collect all the common features of their categories. Classes intended for use in CODER systems are indicated by an C after their class names in the inheritance diagram; classes for MARIAN by an M. Classes without question marks in these annotations are the most necessary, and should be implemented first.
5. The drawings of windows attempt to conform to the standards for Motif windows in the X system. Since only one of the authors has any familiarity with Motif, some standards have undoubtedly been violated. Implementors should be aware that this is unintentional, and are encouraged to correct such oversights. Similarly, no attempt should be made to duplicate the appearance of the windows or window components where it runs contrary to accepted or practice or idiom. (For instance, none of the scroll bars have slides. This is strictly an oversight; we expect implemented scroll bars to have whatever slides are common in the idiom.) Finally, all of the windows are approximately the desired shape and size (relative to each other). Implemented windows should begin their life in a shape and (relative) size comparable to that shown. Minor changes to shape and size, however, are expected in any real product.

Document History

This document began in early 1991 as a set of mock-ups for an X-Window interface to the INCARD system, produced by P. Koushik. It was intergrated with PK's *Functional Specifications for CODER User Interface Managers* and expanded to include lexical uses of CODER by Robert. During the spring of 1991 it was substantially revised and added to by Robert and PK, with much assistance at spotting and filling gaps by the current team of CODER user interface implementors: Ajit Naidu, Nasser Ghazi, and Siva Challa. In the summer of 1991, Robert further re-organized it and expanded it, and in the fall of that year adapted it to service the MARIAN system. Ben Cline, Tim Rhodes, and Eskinder Sahle have all contributed to that process.

V. 2.3 11 Sept. 91: This is the first "released" version. All previous versions were labelled "drafts" and should be considered such.

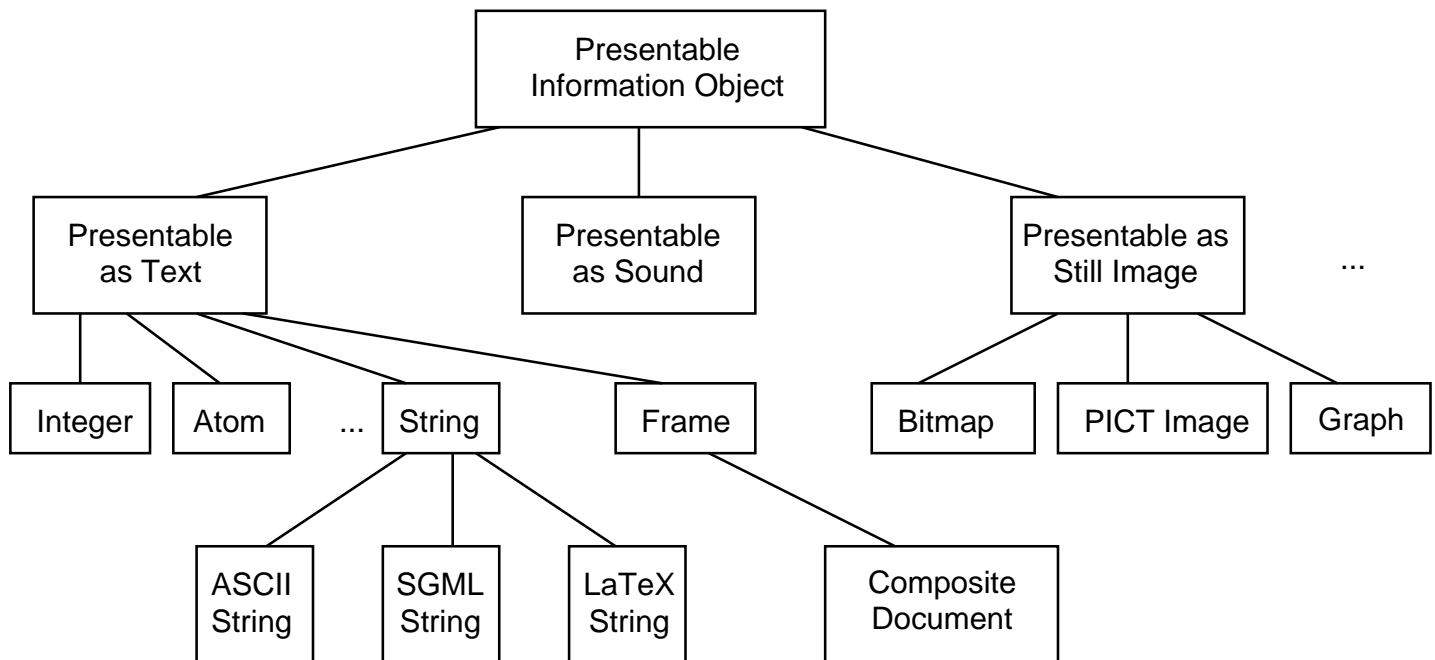
- V. 2.4 27 Jan. 92: Major changes to the structure of the document: a table of contents and introduction were added, as well as page numbers in the object inheritance diagrams. Some "window" talk was changed to "object" talk to better suit non-window implementations. The "Background" class was renamed "Base" and removed from the Modal Object hierarchy, where it really only belongs in X-Windows implementations. The only CLASSES changed were the Bibliographic Form Query Creation class and the Retrieval Set Display class, although minor errors were corrected in the User Identification and MeSH Node Display classes.
- V. 2.5 24 Aug. 92: The interaction object hierarchy was completed by rooting it in a class of "Interaction Objects". All functionality common to all objects in this system was removed to this class. In a few cases where object names were missing from additional methods (notably clear() and goAway()) or callbacks (notably sendHelp()), they were added. Default arguments were added to the biblioQueryCreation class, so that old queries may be summoned up and edited. In addition, an argument was added to the callback for that class to suggest a maximum number of retrievals to present in the first batch. Minor changes were made to menus in the the base object and query creation object classes. And as always, explanatory text was added and revised.

General Notes

1. We make the standard distinction between "modal" and "non-modal" objects: a modal object is one which demands the user's attention by making it impossible for hi/r to perform any action until the object has been dealt with. These are intended to be used sparingly, and so are limited to information and warning transactions and to the initial login dialog. Most objects, and most classes of objects, are non-modal: they permit the user to ignore them while performing other actions. Both modal and non-modal objects operate asynchronously with the underlying information system (CODER or MARIAN), permitting the user to request help or (in the non-modal case) to work with other transactions before returning to supply the information requested in the object or choose an option that the object offers. All user actions are signalled by "callbacks:" additional C functions that pass information back to the underlying information system.
2. All interaction object classes provide some way for the user to get help; either by buttons or by menu selections. **Help** actions always result in processing outside of the user interface, in CODER or MARIAN proper, to determine the correct response. Generally, this will involve further calls from CODER or MARIAN to the interface to display non-modal windows with scrolling text in them, using the showText() window function. Since these further windows also have help provisions, nested **Help** actions may occur.
3. **Inheritance** among classes occurs in two ways. First, most subclasses are formed through functional enrichment, either as an increase in the number or specificity of the methods presented to the underlying information system, as an increase in the number or subtlety of the actions that may be performed by a user, or – most commonly, as these two are different aspects of the same thing – as an increase in both. Second, several child classes are constructed out of combinations of parent classes. Objects of such classes are composite objects, each of the parts of which can be recognized as objects of the parent classes.

Methods and user actions of parent classes are always inherited by their children – there is no defeating of inherited values – so methods, menu items, and so forth are not duplicated in the descriptions of the window classes except where necessary to avoid confusion.
4. Within each object category – the modal and the non-modal – the class hierarchy has been set up to follow the hierarchy of the data objects being manipulated. Not surprisingly, the richest branches of the hierarchies deal with the class of text objects, its subclasses, and classes of composite objects built from text objects. Little or no attempt is made to deal with other primitive data objects such as integers or atoms, as it is assumed that each class of primitives includes a morphism to map its members into unique string representations, such as decimal representations of integers or names of atoms.

To be specific, we establish two hierarchies: a hierarchy of presentable information objects, which are the objects that the system may store and the user experience, and a hierarchy of objects used as carriers of presented information. A portion of the first hierarchy is shown below.



Information objects that can be presented as simple text include text objects themselves, text objects with formatting controls, and other atomic objects such as integers and atoms where a canonical mapping into text has been defined. It is also possible to present some classes of composite objects as text. For instance, it is fairly common to present a frame object or a record object as a block of text, using formatting control to distinguish the slots or fields:

Title: Beginning model theory
Author: Bridge, Jane
Imprint: Publisher: Cambridge University Press
Year: 1979
Subject: Model theory

Other presentations are also possible. The bibliographicFormQuery class in this document, for instance, takes a different approach to the same type of object.

Some information objects are not presentable as text, but may be presented using sound, still images, video and so forth. This document does not cover interactions involving these objects, but could be extended to do so. The objects would look different, but the use of objects to carry information and context, and to guide user actions, would be the same.

Some information objects, in contrast, cannot be carried well by any single presentation object. One important class is the navigable graphs used in hypermedia. Unlike graphs that can be adequately presented by a single still image, hypermedia networks must give the user the means to move between nodes and among regions of the graph; to expand, contract, and reorganize his/her field of view. Much of this navigation requires graphic support for visualization and manipulation. We have, however, been able to provide some support for hypermedia using a composite of text objects to represent hypermedia nodes. Other classes require a collection of objects in order to represent a set or list: a list of choices, a set of retrieved documents, a collection of chapters in a book. An entire hypermedia network, to combine the two classes, can be considered to be a set of hypernodes.

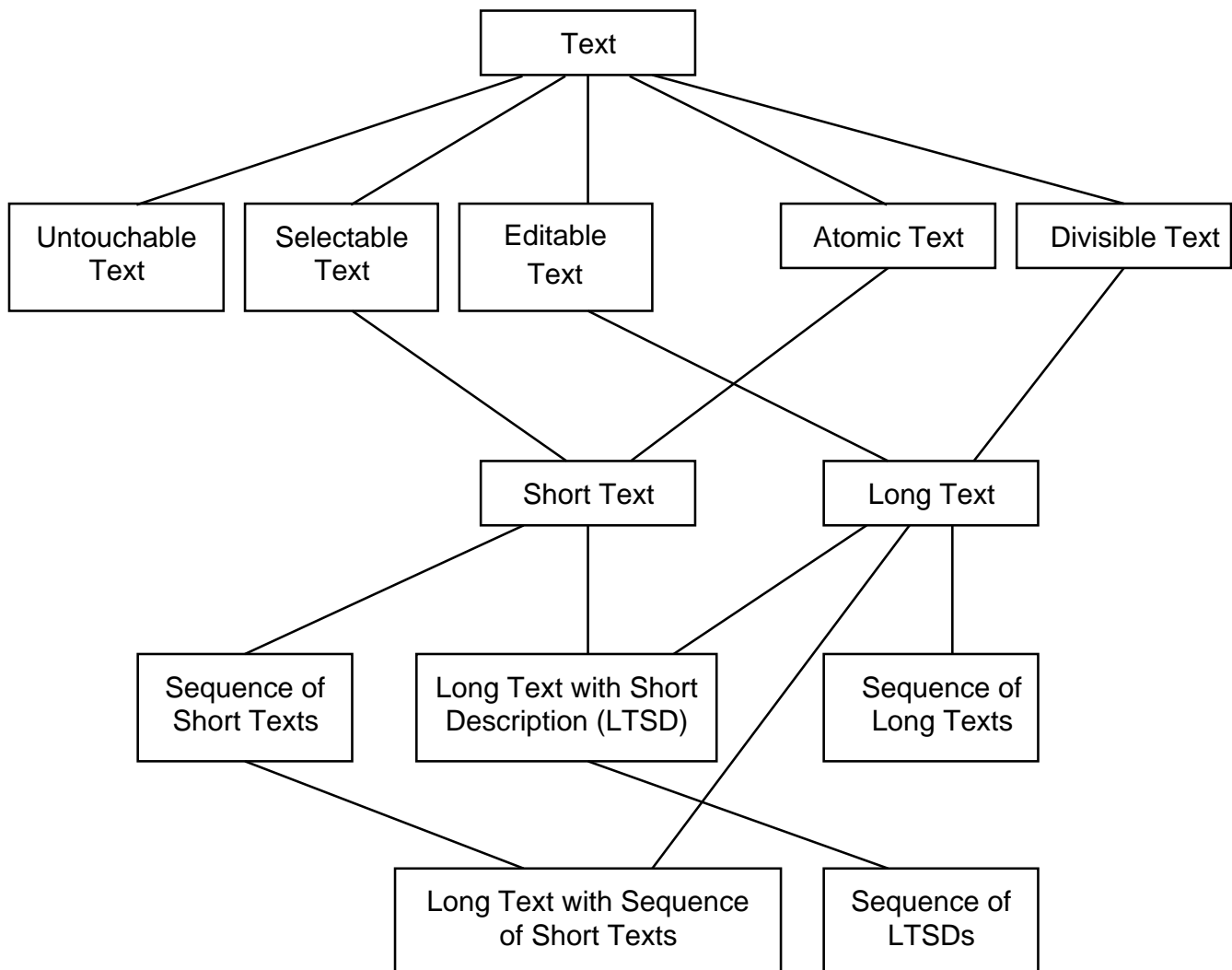
To represent this variety of information object classes: atomic objects, structured objects, composite objects and collections of objects, we must use similar techniques to create and combine the presentation classes used in constructing our interface objects. Take as an example the text objects used in building the interaction objects in this document. We can create several different kinds of text fields in our interfaces. On the one hand, we can create text that the user can or cannot interact with. We distinguish three subclasses of text here. There is *untouchable text*, used for the titles of objects and the labels on object parts. The user cannot interact with this text at all, except to read it. There is *selectable text*, which the user can select but cannot alter. Selectable text allows the user to call attention to a part of a field; for example, it may support a **Copy** operation in editing. Finally, there is *editable text* which the user may freely alter.

An orthogonal division separates *atomic text* from *divisible text*. Atomic text may only be manipulated as a unit; divisible text may be manipulated in smaller units. Common examples of atomic text occur in menu items or (in some graphics tools) labels in graphs. Divisible text is most familiar to us through word processors: editable divisible text is the norm in most interaction objects. The atomic/divisible distinction does not apply to untouchable text, but makes an important difference in both selectable and editable text fields. This system of interface objects makes heavy use of two such categories: the short atomic selectable text and the long divisible editable text (see diagram next page).

From these descendants of the text class we construct others through composition and collection. Each interface object that we build uses these three techniques, of specialization, composition and collection to create ever more complex classes of interaction objects from the same basic building blocks.

These classes then become our tools for representation of information objects (see diagram in two pages). Primitive objects like atoms and integers are represented by short pieces of atomic text. Primitive string objects may be represented by different sorts of primitive text objects in different presentation contexts. We can use a collection of short texts to represent a list of alternatives, so that the system may dynamically construct lists from which the user may make choices. A collection of long texts can represent a book, either with each member of the collection predetermined by an analysis into chapters and sections, or with each member being dynamically created by the user as s/he places "bookmarks" in the text. Selectable atomic texts are useful as titles, both in the conventional sense of the title of a text document and as the titles of fields in structured objects. The interaction objects in this document comprise many variations on this theme.

Text Field Classes

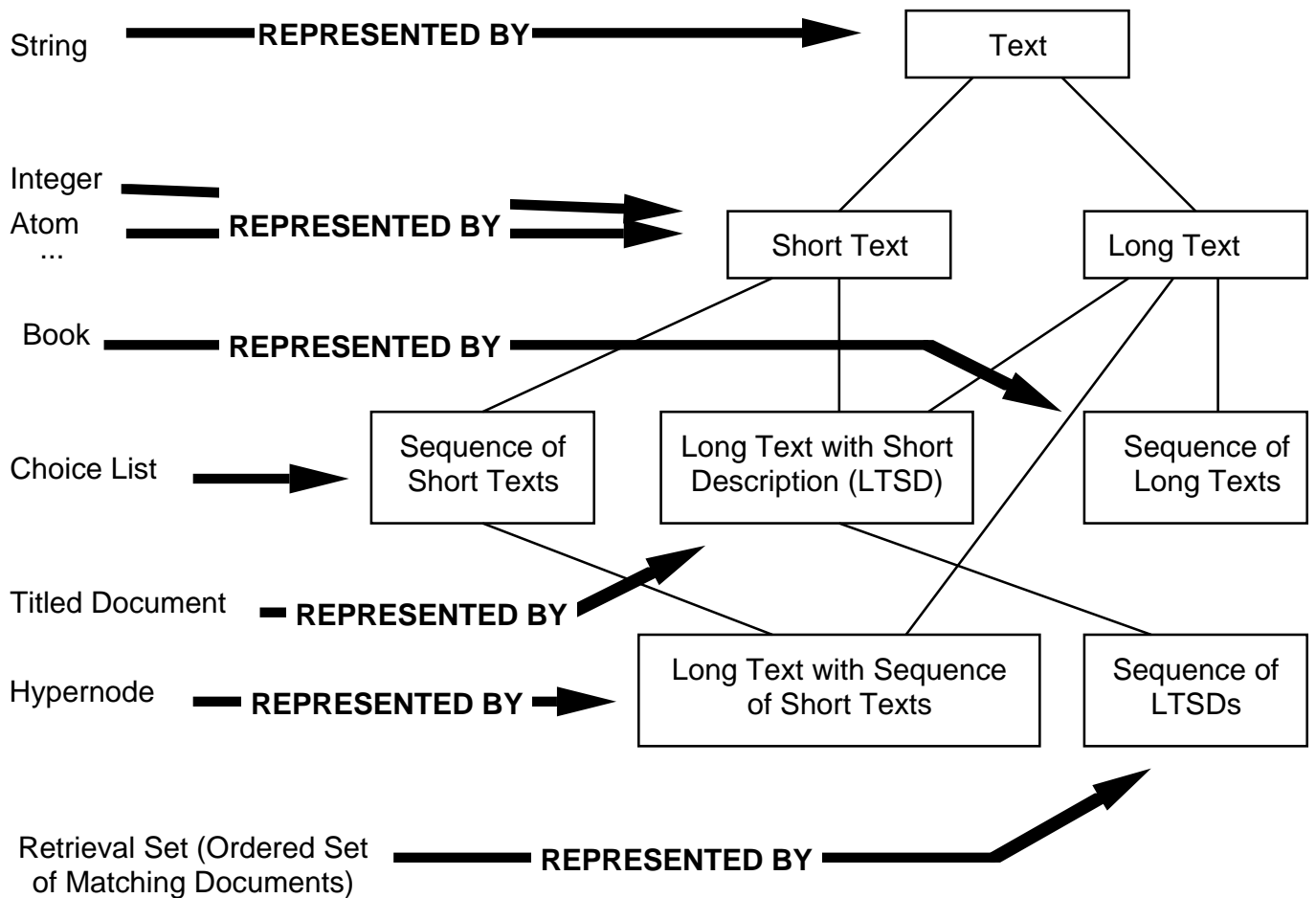


Two complex examples are worth mentioning. As mentioned above, we treat hypermedia networks as collection of hypernodes, where each node has both information content and links to other nodes in the network. When this information content can be represented as text, we refer to these as *text-valued hypernodes*, or simply text nodes. Each text node is composed of a text object representing the nodal information, generally as a long editable text, and a sequence of selectable atomic texts, each representing a link. These short texts are constructed by the underlying information system, generally from the label on the link together with a short description of the target node. A collection of these textNode interface objects then serves as a viewport into the hypertext network.

The second example is the ordered set of documents retrieved by a query. This set must be presented to the user in such a way as to make it possible to scan its members quickly, either in the order recommended by the information system or in random order. To achieve this we use a collection of long texts (the documents retrieved) with short descriptions, the whole presented as a single interaction object.

Information Object

Interaction Object



Following these principles has made it possible to organize an extremely complex set of requirements into what we hope is a minimal system for achieving the broad functionality required by advanced retrieval systems like CODER and MARIAN. We hope that understanding the principles will also make it possible for the reader to understand how the set is organized and how it can cover the needs of such systems. Finally, we hope that following the hierarchic nature of the windows and the presentation objects out of which they are constructed will make the implementor's job easier, and the number and complexity of the interaction object classes in this document less forbidding.

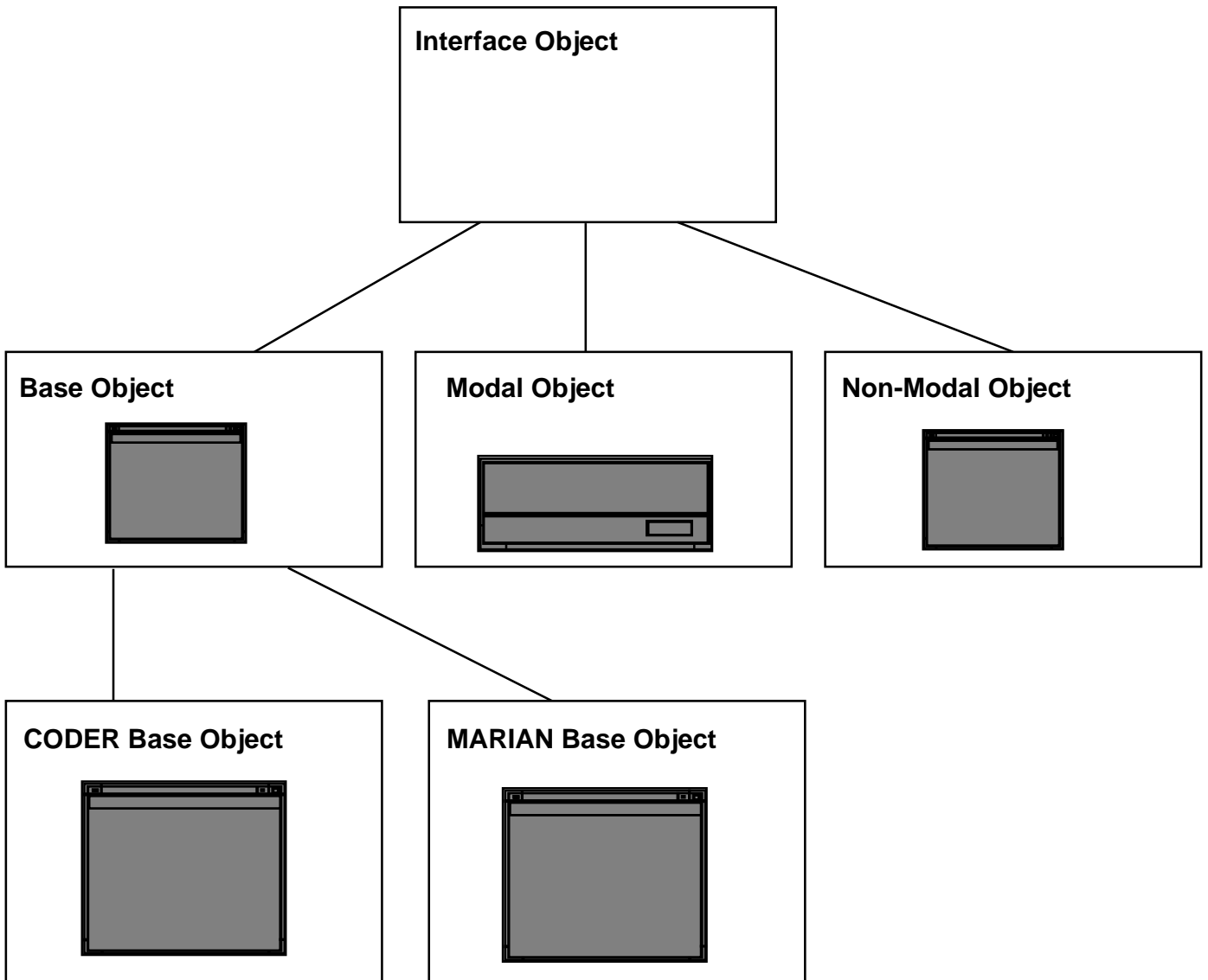
Implementation Notes

1. All functions, both window functions and callbacks, are to be written as integer-valued functions that return 0 iff their action has been successfully performed. Window functions should return negative values if they fail; some standard values are listed at the end of "Writing Modules for the CODER/F3L Environment". Return values of the callbacks are to be tested, and some appropriate action taken on a non-zero return, typically signalling the user. All subroutines and system calls should be similarly tested, to avoid exceptional conditions. Under no conditions should any of the functions exit without return ("bomb"; "die") or return a 0 if they have not been successful.
2. At the time of this writing, all text displayed in any window class is straight ASCII text. Plans are already under way, though, to include format, font and style control. Immediate expectations are for centered text, bold-faced text, and pre-selected text (probably using reverse video). Development should not be stalled waiting for exact specifications of how this should work. Rather if there are widgets available that permit text with some or all of these attributes, they should be used, and the representation for formatting and style documented. This applies both to what the user would perceive as text fields in windows, and to the label, title and message fields that the user cannot access, but that are determined at window creation time by method arguments.
3. Most of the displayed text contains ASCII tab and newline characters, which are to be interpreted as usual. This is not always the case, though -- notable exceptions are dictionary text and some of the text displayed in showText() windows. These texts can run for indefinitely long times without any formatting control: texts over a thousand characters "broad" are not uncommon. Two approaches can be taken with such text: either the text can be broken on word boundaries into lines of reasonable length (say, between 70 and 80 characters), or the text can be left on a single line and reached through horizontal scrolling. Either approach is acceptable, but if the second is taken, you should remember that there is no theoretical limit to the size of the line that may result.

On a related note, titles for windows and other "untouchable" text fields may run up to 256 characters. Space should be left for the maximum possible, and if a title must be split into lines to fit into the space, it should also be broken between words. In later versions, we may include format control in titles for such features as centered text, font control and so forth. If means are available through some idiomatic tool for this, please use it and document how such formatting can be achieved.

4. Scroll bars are generally shown, but should not appear to the user until there is enough text in a window to justify them. Then they should always appear. All text fields, and all choice list fields, should include both horizontal and vertical scroll bars.

Interface Objects



Interface Object

General Notes

There is a line in the Tao Te Ching, "the perfect square has no corners." Similar paradoxes abound in object-oriented programming. Here, for instance, the parent interaction object has no realization. It is a screen with no format; a window with neither size nor shape. It is in some sense raw functionality; the encapsulation of the functions that all interaction objects must present to user and system. To the system, it presents the capability of any object class: creation and destruction of objects. To the user, it presents the abilities that we guarantee the user to have in any situation: to quit the session, to review system status and history, and to seek help. In addition, this class defines the editing capacity of the system, even though its virtual objects have no editable fields. This ensures that the editable fields of all actual objects, when defined, will behave in the same way, and that material cut or copied from one will be able to be pasted into another.

Each class of interaction objects supports a call that creates a new object of that class. Due to the architecture of the user interface manager that mediates the classes, and the fact that it cannot be assumed that the interface manager will be implemented in an object-oriented language, the constructor method has a different name for each class. In this document, the constructor is detailed at the beginning of the description of each interface object class. The name and signature of the destructor method, though, is constant:

```
int
goAway(+ObjectName)
char* ObjectName;          /* Title of a non-modal or message of a modal object. */
```

A unique object name is part of all calls to interface objects in this system. This name is specified as part of every constructor. Constructors called with the name of an existing object are assumed to "re-construct" the object: to return the object to its initial state and replace all specified features with their new values. For example, calling the constructor showTextObj() on an existing textDisplay object will cause the text of the existing object to be replaced by the text in the new call. The object name is used in any methods, such as goAway(), that are supported by created objects; it is also used by the callbacks that signal user manipulation of an object to identify which object has been affected. In the case of objects with editable fields, for instance, a uniform method:

```
int
clear(+ObjectName)
char* ObjectName;          /* Title of a non-modal or message of a modal object. */
```

provides the system with the capability of resetting all editable fields to null. Calling one of these "additional methods" with an object name that is not associated with an extant object has no effect on the user interface process, but will result in a non-zero return value for the method.

At any time in the system/user dialog, the user has certain inalienable abilities: the ability to ask for help, to review the status of the dialog, and to quit. How these abilities are provided varies both among implementations and among the classes of windows. For example, **Help** is regarded as an atomic action in a modal object, whereas in non-modal objects it involves a choice of subjects, and is thus generally implemented as a menu. However it appears, though, it is present in every case, and it produces the same callback to the underlying system:

```
int
sendHelp(+ObjectName, +ItemName)
char* ObjectName;          /* Title of a non-modal or message of a modal object. */
char* ItemName;           /* Class name for modal objects; subject for non-modal. */
```

Similarly, it is always possible for the user to quit, although this may require different actions under different circumstances. In the NeXTStep implementation, for instance, **Quit** is provided for modal objects by allowing the user to access the Base menu while the modal object is on screen; while other implementations require it to be a button on the object itself. **Quit** triggers both whatever local processing is needed to shut down the user interface and a callback telling the underlying system to shut down. The callback is a special instance of a more general call:

```
int
endSession(+Condition)
int Condition;             /* 0==NORMAL (user chose "Quit" from menu). */
                          /* Other values determined by user interface manager code. */
```

Status and history enquiries are user actions that are still under development at this writing (v. 2.5). We believe that these are an important powers to give the user, and that once given, they should be globally available. Consequently, this is the appropriate place to define them. They just haven't been defined yet. MARIAN supports through a menu in the base object, a callback:

```
int
status(+Title)
char* Title;              /* The title of the base object; thus presumably of the application. */
```

This call invokes processing by the MARIAN Session Manager that presumably eventually results in a text object being synthesized and displayed to the user. How adequate this is, and whether any sort of history facility can be added to it, depends largely on the function of the underlying system. Further exploration will doubtless yield refinements.

In interaction objecty classes where editable fields are present, the usual three functions, **Cut**, **Copy**, and **Paste**, are provided for editing. Text or images cut or copied from any object is saved in a single buffer for pasting into any editable field in that or any other object, whether modal or non-modal. In fact, although modal objects have no menus by which to invoke these functions, any "short cut" versions of editing commands should work even when a modal object has control of the interface. The actual implementation of cutting, copying, and pasting are handled by the user interface manager, and thus produce no callbacks. The edit functions are defined here, at the top level of the hierarchy, but are not uniformly available throughout the hierarchy. In particular, there is nothing to edit in either the parent window or the Base windows, so the functions should be either absent or disabled. Certain other classes of windows allow copying from, but not alteration of the text displayed.

Base Object

General Notes

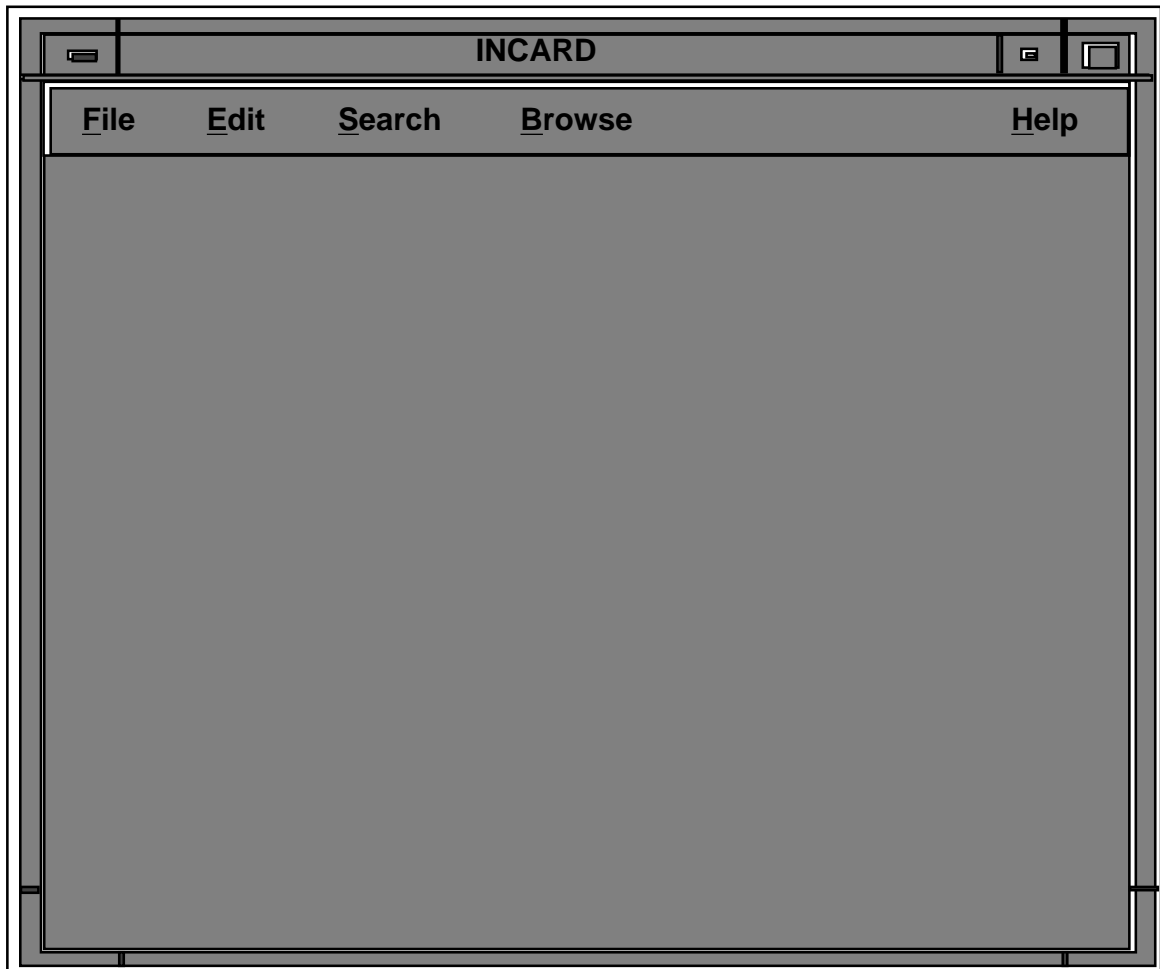
This is the class that varies most between different implementations. Each implementation style has an idiomatic way to deal with the basic functionality of an application, and how to present it to the user. In a menu-based system, this functionality is typically encapsulated in the "Main Menu," sometimes with the addition of function keys and the like to short-circuit the flow of menus. In the Macintosh and other PARC-descended systems, the functionality is represented by the (unique) "Menu Bar." The NeXT has a slightly different idiom, involving an "Application Menu" displayed usually in the upper left-hand corner of the screen. In X-Window systems, basic functionality is encapsulated in a "Background Window." And so forth.

In any system, though, there is some way of executing various functions that radically change the dialog between user and system: starting and stopping dialogs; quitting the application; getting global help and checking system status, and so forth. These functions are assigned in this system to the Base Object class. The representative Base objects that follow are shown as X-style background windows. If you are implementing these in a different idiom, do not be confused: the window is not important; the functions are.

CODER / INCARD Base Object **

```
int  
showBase(+Title)  
char *Title;
```

e.g.: showBase("INCARD")



NOTES:

Ideally, the contents of the **Search**, **Browse**, and **Help** menus should all be passed into this function as parameters of type `char * []`. Since dynamic menus appear very difficult in X, we are opting for this static version. This window, and all other non-modal windows, should nonetheless be coded so that the menu contents should be easily changed in the code, if not at run time.

MENUS:

Menu	Item	Action
File	Quit	
Edit	Cut Copy Paste	
Search	MeSH Thesaurus Medlars Collection Cardiology Course Notes Collins English Dictionary All	callback: searchCollection()
Browse	MeSH Thesaurus – Hierarchically Medlars Collection – By Author Medlars Collection – By Journal Cardiology Course Notes – Table of Contents Cardiology Course Notes – Index	callback: browseCollection()
Help	On CODER On Searching On Browsing	

CALLBACKS:

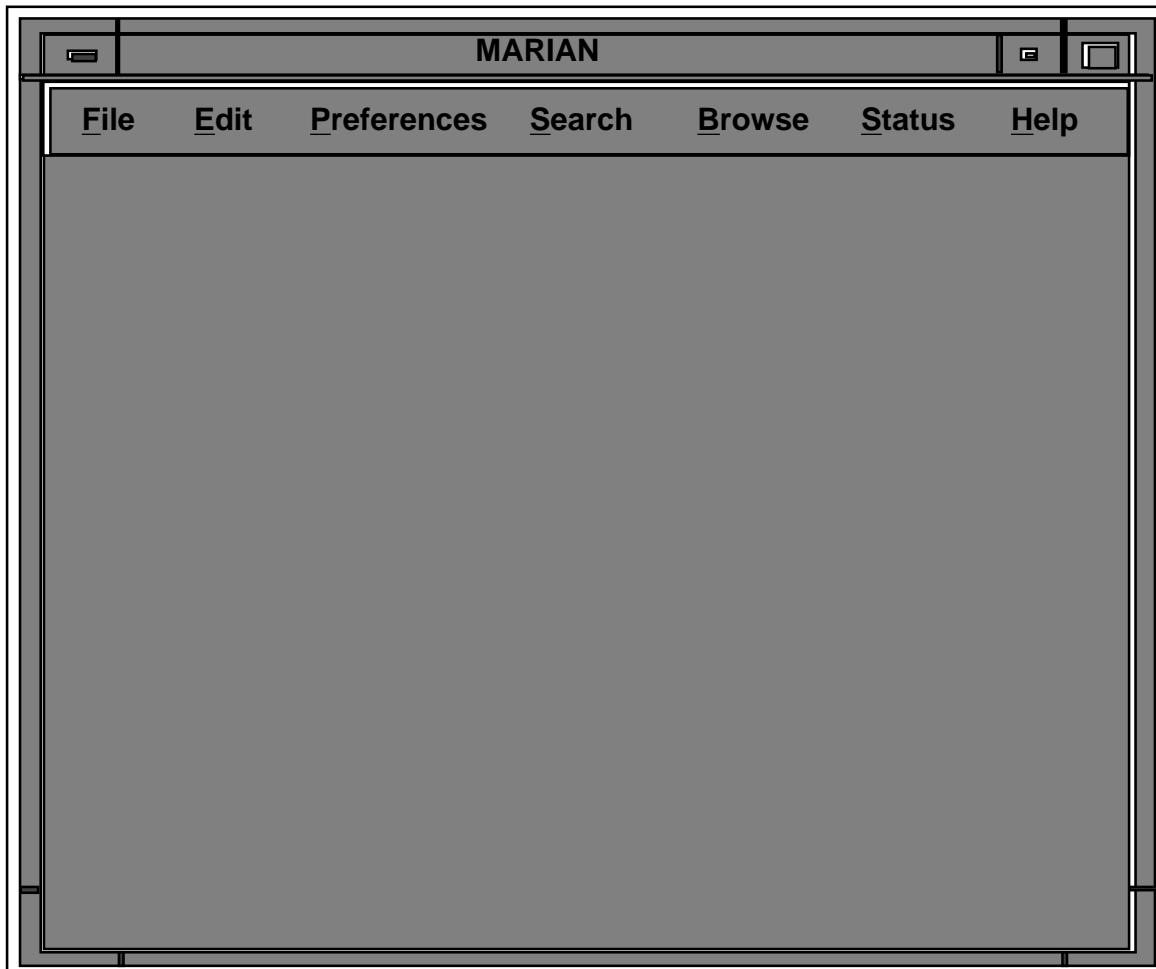
```
int
searchCollection(+ItemName)
char *ItemName;
```

```
int
browseCollection(+ItemName)
char *ItemName;
```

MARIAN Base Object **

```
int  
showBase(+Title)  
char *Title;
```

e.g.: showBase("MARIAN")



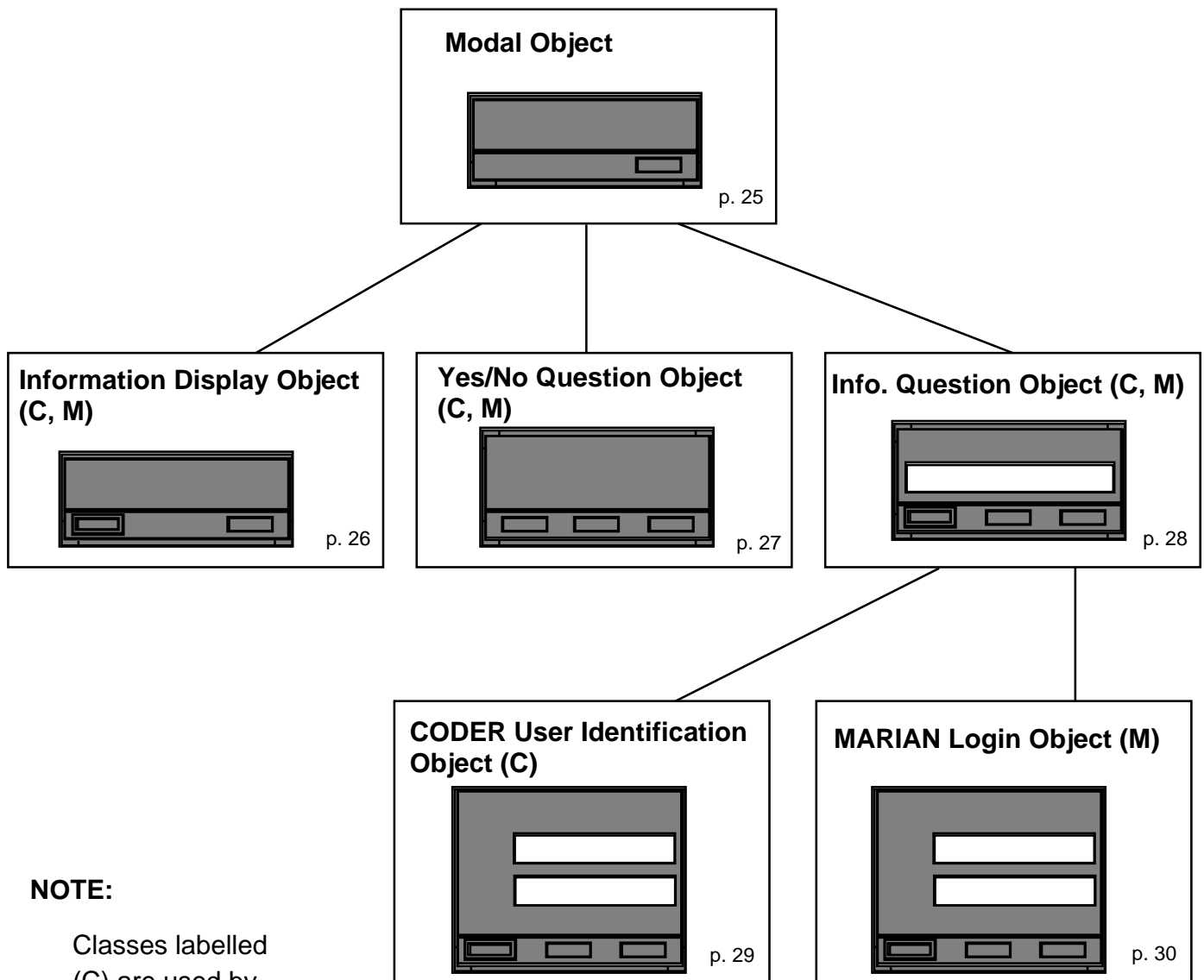
NOTES:

This class differs from the `incardBaseObject` class only in the options in the **S**earch, **B**rowse, and **H**elp menus. See the note for the preceding class.

MENUS:

Menu	Item	Action
File	Quit	
Edit	Cut Copy Paste	
	Preferences	
Search	VT Catalog	callback: searchCollection() – see preceding class
Browse	By call number By cluster Author list Subject list	callback: browseCollection() – see preceding class
Status	Show session status	callback: status() – inherited from parent
Help	On MARIAN On Searching On Browsing	

Modal Objects



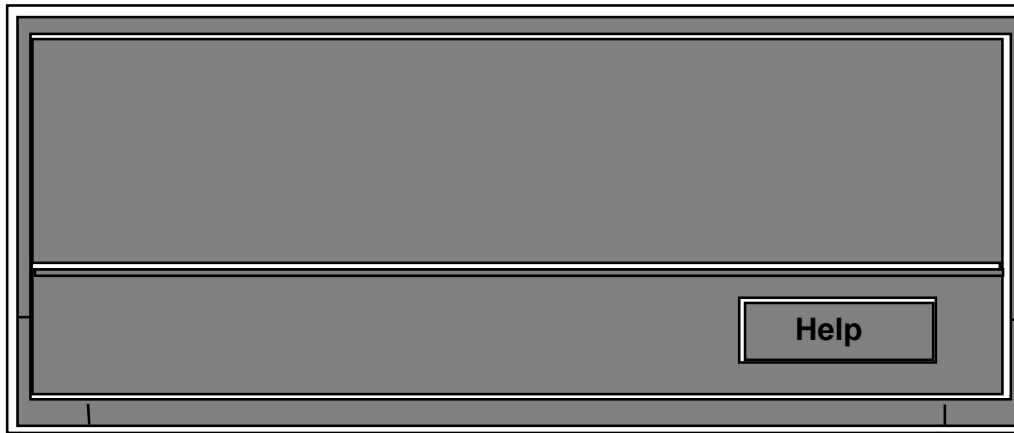
NOTE:

Classes labelled (C) are used by CODER; those labelled (M) by MARIAN.

General Notes

A modal object claims the user's attention: It appears "in the foreground" and will not relinquish the user's attention until it has been dealt with. In a windowing system, this means that a modal window appears "in front of" the other windows on the screen, and is the only window on the screen that the user can manipulate until it has been sent away. In a menu system, a modal object appears as an interaction that occurs as the user leaves one object for another before the other appears. Having one modal object presented to the user does not freeze the system, however: the system is operating independently, and may send other messages to the interface while the user is dealing with the modal object. In that case, the interface can queue the messages either in or out of the user's field of perception. IN particular, the system can send messages to construct several modal objects of the same class. The interface manager may show all these to the user simultaneously or sequentially, but as far as the system is concerned, the interface is dealing with several modal objects at once. Which is which is determined by the system-generated message associated with the object. Thus, all callbacks from modal objects include the message text as a determiner of which object they encode a user's response to.

All modal objects have a user function for acceptance of their associated action, usually labeled **Done**. Most also give the user the ability to reject or cancel the action, usually labeled **Dismiss**. Both of these buttons "send the object away," thus causing the initiative of the dialog to return to the user. In all cases a function labeled **Help** is used to indicate confusion or to request aid. Activating this function does not send the object away, but initiates the display of a "help" text forwarded by CODER in a separate interaction. This help object along with any further help objects if spawns should be like a non-modal object in that the user may move and re-size it, keep it or make it go away. The original modal object, though, should remain active throughout the user's actions with the help interaction, and no other object should be available to the user until some user function on the modal object is activated. At that time, both the modal object and all its generated help objects should go away.

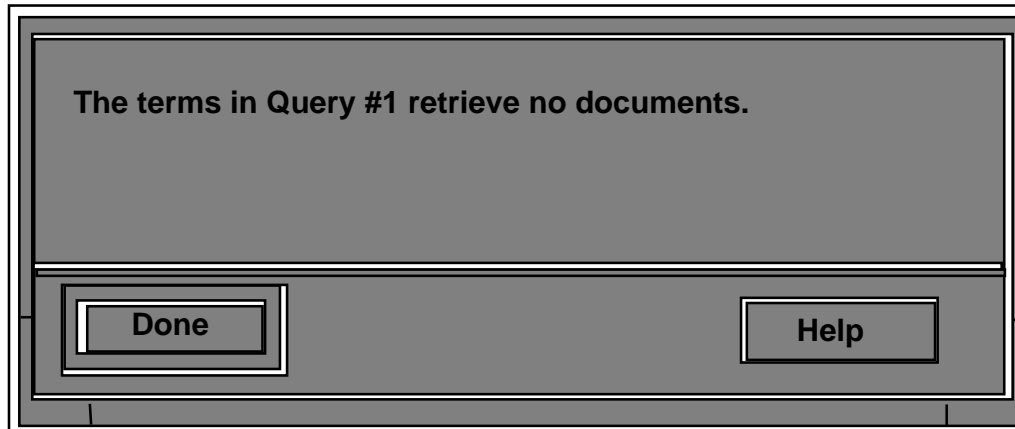


CALLBACKS:

```
int  
sendHelp(+Message, +ClassName)  
char *Message;          /* The text of the window message (all actual modal objects */  
                          /* contain a unique message field.) */  
char* ClassName;       /* The class of modal object where the call originates. */
```

```
int  
showInformation(+Message)  
char *Message;
```

e.g.: showInformation("The terms in Query #1 retrieve no documents.")



NOTES:

If there is a standard "information" icon in X (in the Mac it is a person speaking) it should be included to the left of the text, and the size of the window determined accordingly.

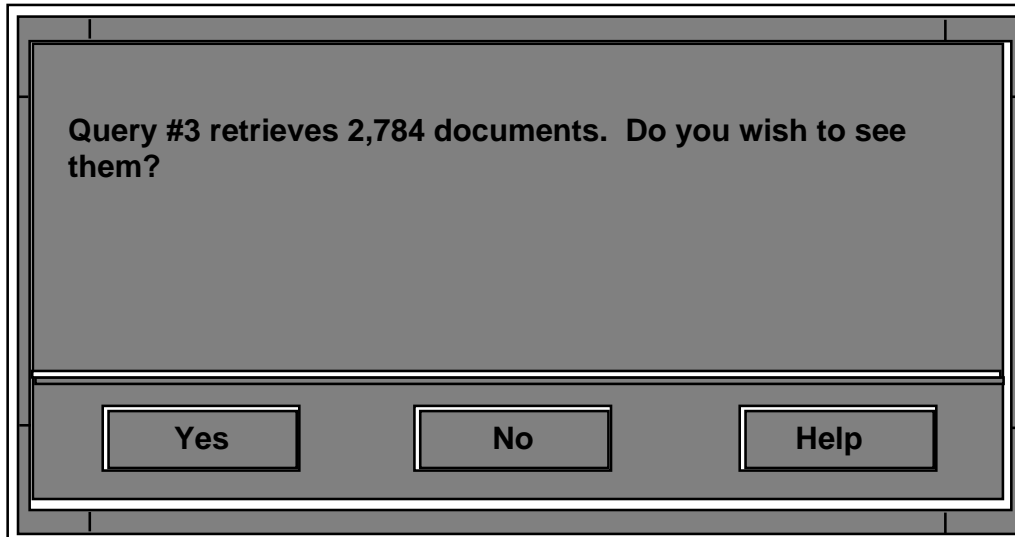
No callback need be made if **Done** is selected. The sendHelp() callback is inherited from the parent.

Yes/No Question Object

27

```
int
askYesNoQuestion(+Message)
char *Message;
char *Response;    /* Space allocated by caller for up to 256 characters. */
```

e.g.: askYesNoQuestion("Query #3 retrieves 2,784 documents. Do you wish to see them?")



NOTES:

If there is a standard "Question" icon in X (in the Mac it is a person with a question mark balloon) it should be included to the left of the text, and the size of the window determined accordingly.

CALLBACKS:

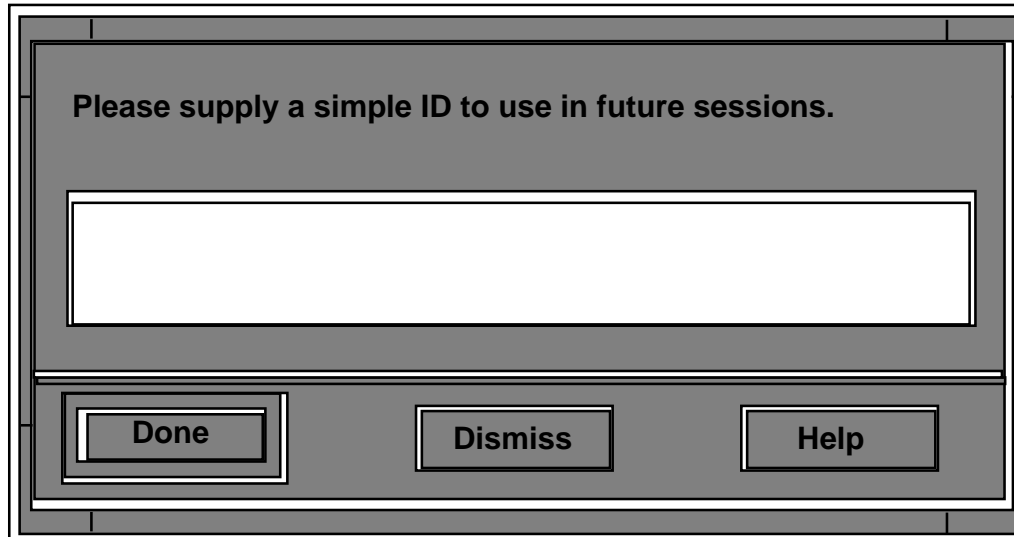
```
int
yesNoAnswer(+Message, +Ans)
char *Message;    /* The message in the object, to identify what question is being */
int  Ans;        /* answered. Which answer was chosen: 0='Yes'; 1 = 'No'. */
```

Informational Question Object

28

```
int  
askInfoQuestion(+Message)  
char *Message;
```

e.g.: askInfoQuestion("Please supply a simple ID to use in future sessions.")



CALLBACKS:

```
int  
infoAnswer(+Message, +Ans, +Response)  
char *Message;          /* The message in the object, to identify what question is being */  
int   Ans;              /* answered. Which action was chosen: 0='Done'; 1 = 'Dismiss'. */  
char *Response;        /* The string entered by the user. */
```

CODER / INCARD User Identification Object

29

```
int  
getUserID(+Message)  
char *Message;
```

e.g.: `getUserID("Welcome to CODER. Please enter either your
CODER identifier (if available) or your full name:")`



Welcome to CODER. Please enter either your CODER identifier (if available) or your full name:

CODER ID:

Name:

Done Dismiss Help

NOTES:

This is basically just a double Informational Question object with static labels.

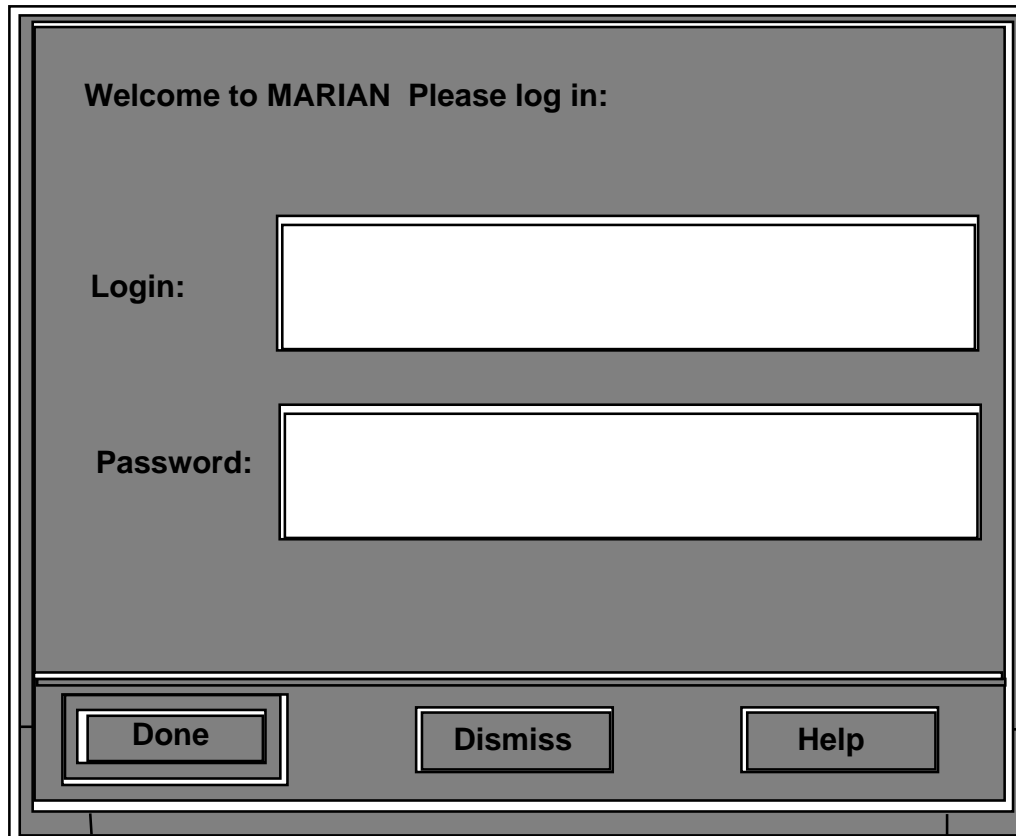
CALLBACKS:

```
int  
userIDAnswer(+Message, +Ans, ID, +Name)  
char *Message; /* The message in the object, to identify what question is being */  
int Ans; /* answered. Which action was chosen: 0='Done'; 1 = 'Dismiss'. */  
char *ID; /* The string(s) entered by the user (NOTE: either or both of */  
char *Name; /* these two may be null. */
```

MARIAN Login Object

```
int  
getLogin(+Message)  
char *Message;
```

e.g.: getLogin("Welcome to MARIAN Please log in:")



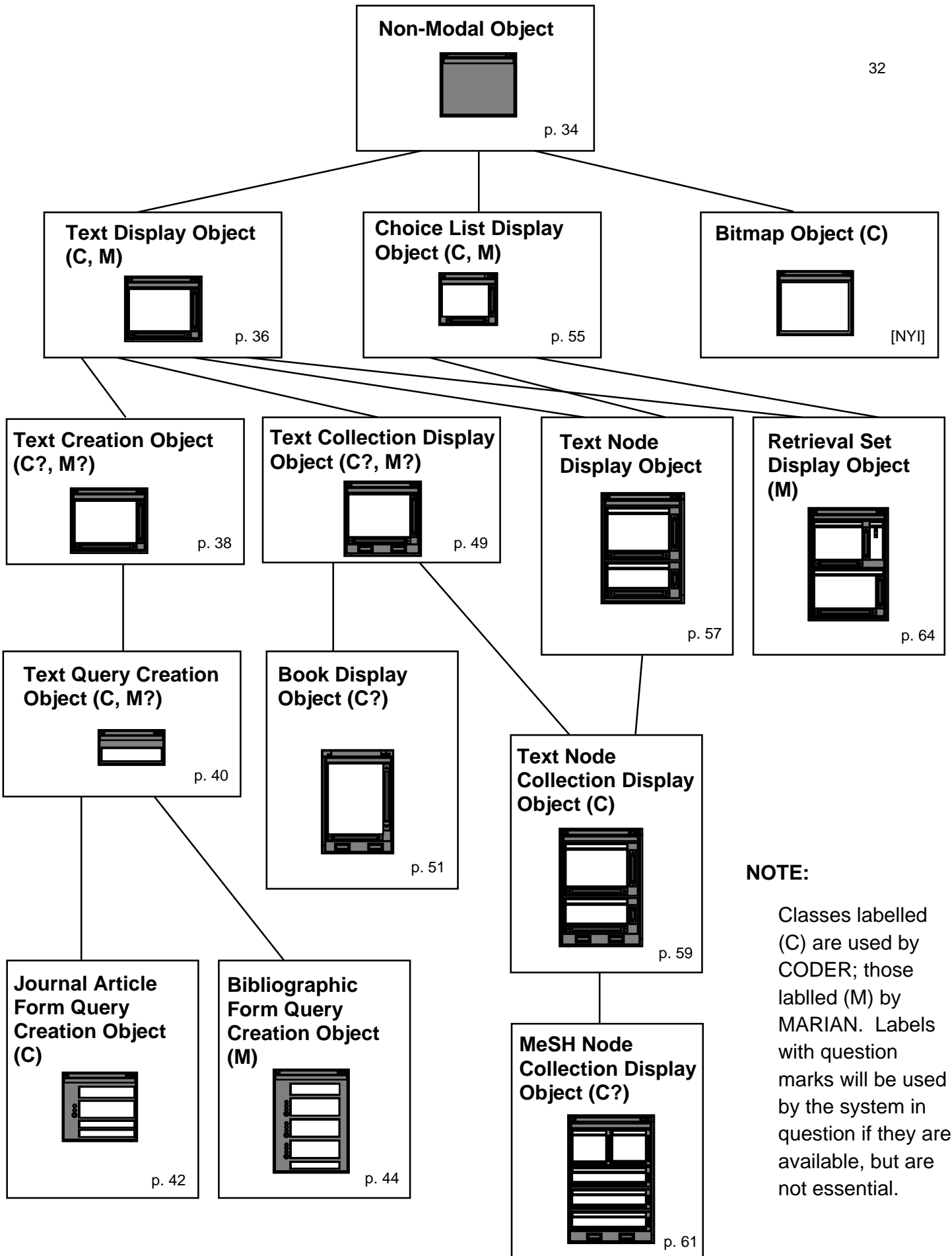
NOTES:

This differs from the CODER/INCARD User Identification object in using a NON-ECHOING text field for the password.

CALLBACKS:

```
int  
userIDAnswer(+Message, +Ans, +Login, +Password)  
char *Message; /* The message in the window, to identify what question is being */  
int Ans; /* answered. Which action was chosen: 0='Done'; 1 = 'Dismiss'. */  
char *Login; /* The string(s) entered by the user (NOTE: either or both of */  
char *Password; /* these two may be null. */
```

Non-Modal Objects



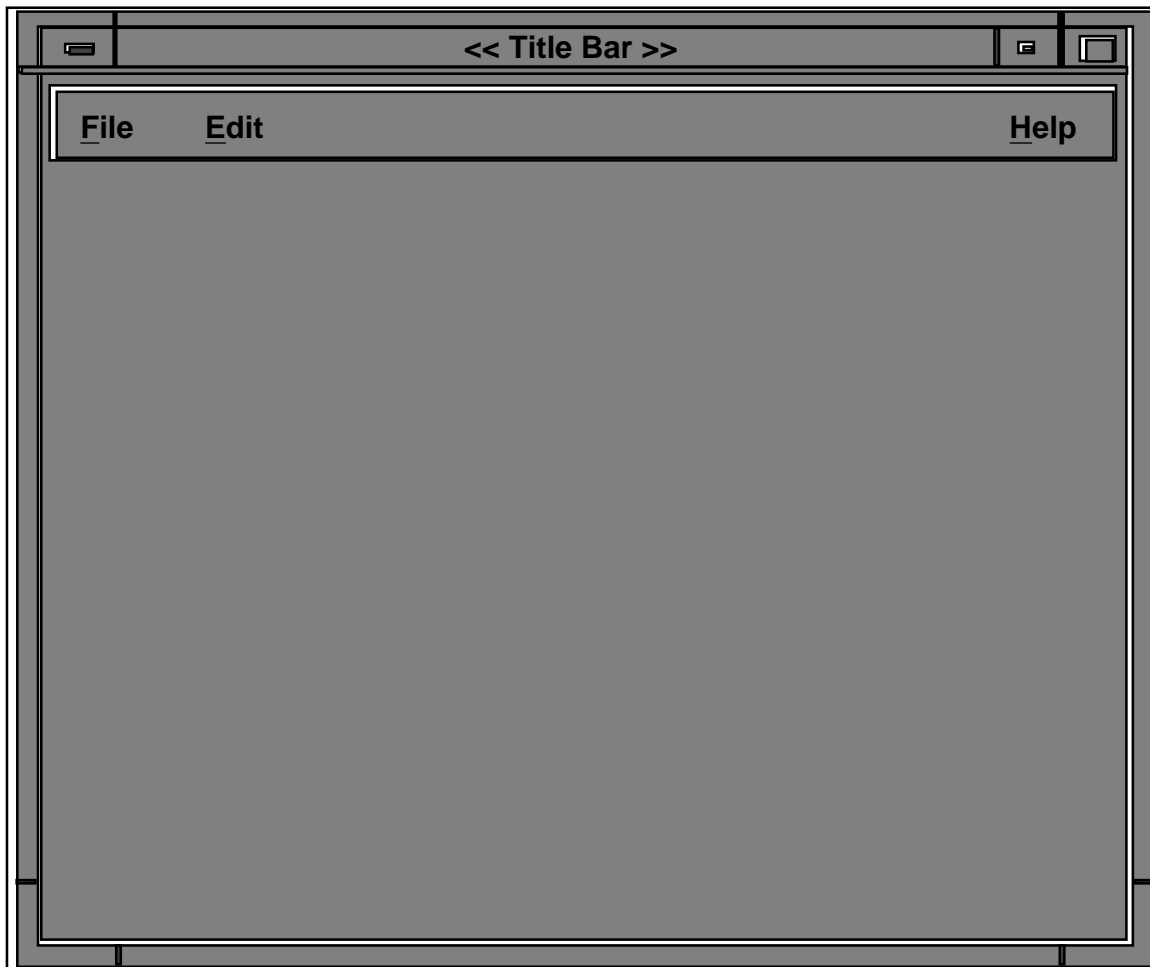
NOTE:

Classes labelled (C) are used by CODER; those labelled (M) by MARIAN. Labels with question marks will be used by the system in question if they are available, but are not essential.

General Notes

Non-modal windows generally use menus rather than buttons for control, although some have buttons as well. In particular, **H**elp is a menu in non-modal windows, with selections determined by the window class. Selecting an item from the **H**elp menu always results in a callback to CODER. Every class of non-modal windows has two other associated menus: **F**ile and **E**dit. The **F**ile menu always has at least one item, **Q**uit, which triggers both a callback to CODER telling it to shut down the system and whatever local processing is needed to shut down the user interface. The **E**dit menu has the usual three functions, **C**ut, **C**opy, and **P**aste. Text cut or copied in any window is saved in a single buffer for pasting into any editable field in that or any other window, whether modal or non-modal. In fact, although modal windows have no menus by which to invoke these functions, any "short cut" versions of editing commands should work even when a modal window has control of the interface. Non-modal windows are summoned by functions that return 0 if and only if the window is created successfully. Some also have associated functions for clearing, augmenting or replacing text fields, or for removing the window at CODER's initiative. All have "go-away" buttons for removing them at the user's initiative, and all can be moved using the standard dragging protocol. Re-sizing should also follow the standard protocol in X.

The edit functions are defined at the top level of the non-modal window hierarchy, but are not uniformly available throughout the hierarchy. In particular, there is nothing to edit in either the parent window or the Background window so the functions should be either absent or disabled.. Certain other classes of windows allow copying from, but not alteration of the text displayed.



NOTES:

No object of this class ever occurs in an actual interface: for one thing, there is no constructor to create a new object of this type. It is included here as a handy place to centralize the functions common to all non-modal objects.

All non-modal interface objects have a title, shown on a title bar or similar prominent place, that serves as the unique name for that object. All non-modal objects have the following capabilities, exemplified in window implementations by the following features: the user can utterly delete the object, in windows by a "go-away" box. The user can remove the object from immediate consideration without destroying it, typically by a pair of "minimize and maximize" boxes. In windowing systems, all non-modal objects can be moved and re-sized, and will remain at their new location and size even while not in view. Further capabilities are usually implemented as a set of menus on a menu bar. This includes the Quit, Help, and editing actions inherited from the parent interface object. Following idiom, we have shown **Quit** as an option of a **File** menu, and have added separate **Edit** and **Help** menus. In order that the Help action be sensitive to the object from which it is called, its callback is parameterized by both window title (object name) and menu item. These menus, together with their listed items and associated actions, are inherited by all descendent classes of non-modal objects. The menus and menu contents are augmented in any actual non-modal class to provide other contextually-determined actions to the user. In most cases, these actions generate callbacks parameterized by the window title and by the menu item selected.

Menu inheritance applies not only to entire menus, but to menu items. For instance, the **Quit** option of the **File** menu is inherited by textCreation objects. These objects also have a **Save** item in that menu; this is added to the **Quit** option, rather than replacing it. Occasionally menus or their items are renamed by descendent classes. When this is done, it is noted explicitly. Unless such a note is made, it should be assumed that any menu or item defined in a parent class is duplicated in the child class.

Nowhere is this agglutination of menu items more noticeable than in the **Help** menu. Beginning from the first item defined in this class, successive descendents each add items, until the most specialized nodes have quite extensive lists of Help options. It will be noted that that first item is really strictly two items, one for use in CODER and one in MARIAN. Thus strictly speaking this class (and all of its descendents) are strictly speaking pairs of sibling classes, differing only in the first item of the **Help** menu. For the most part we will ignore this in this, and all further case of close sibling classes.

MENUS:

Menu	Item	Action
File	Quit	<< Quit processing inherited from parent interface object class >>
Edit	Cut Copy Paste	<< handled by interface manager; inherited from parent >>
Help	On CODER / On MARIAN << Additional contents vary from class to class >>	callback: sendHelp(), specialized form parent.

ADDITIONAL METHODS:

goAway() and clear(), inherited from parent interface object class.

CALLBACKS:

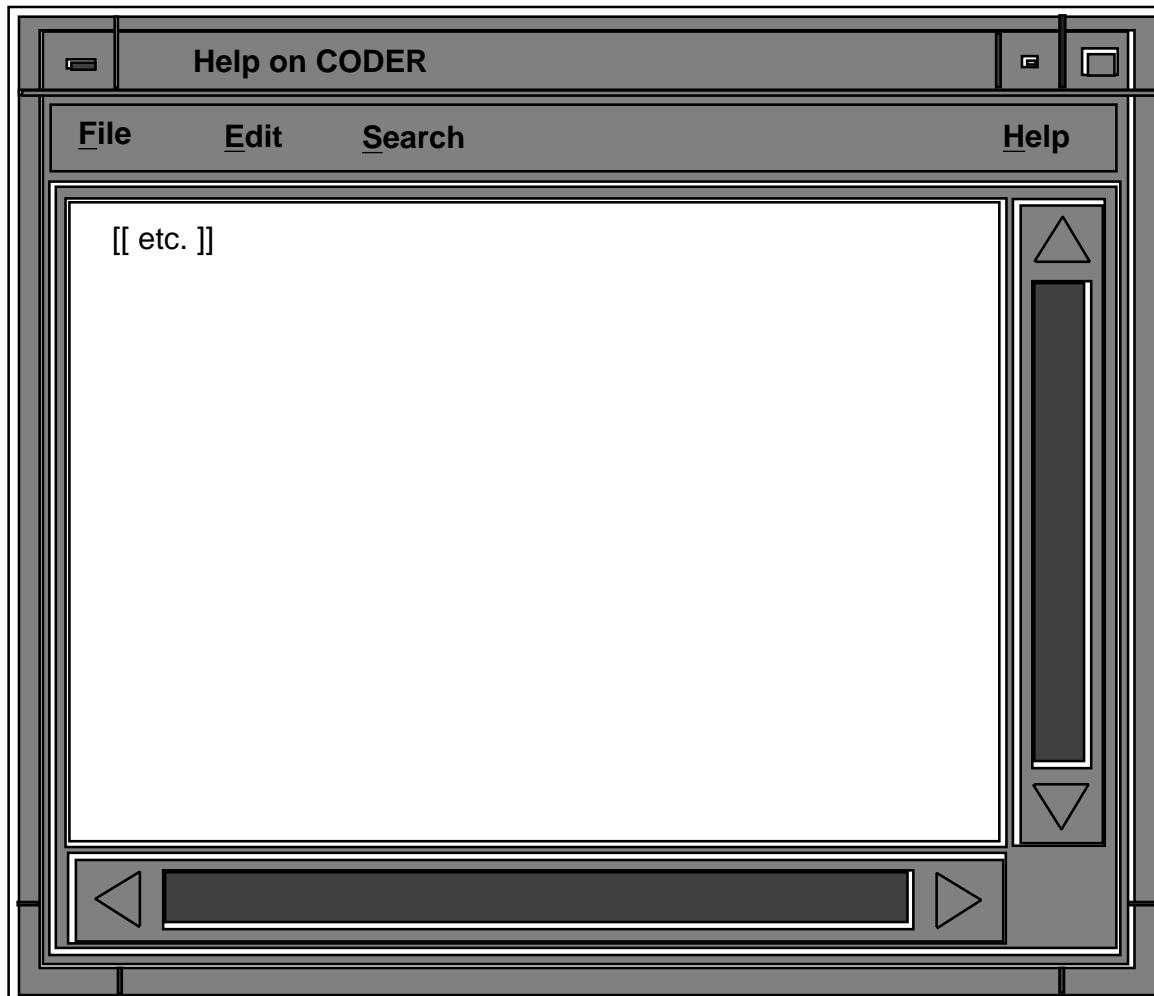
```
int
sendHelp(+Title, +ItemName)
char *Title;
char *ItemName;
```

Text Display Object

36

```
int  
showText(+Title, +Text)  
char *Title;  
char *Text;
```

e.g.: showText("Help on CODER", "[[etc.]] ")



NOTES:

In addition to **Quit**, the **File** menu contains a **Save** option. Selecting this option causes the text to be saved to the user's file space. This is probably accomplished by the user interface manager itself, but we treat it as a callback for consistency.

The **Search** menu includes three items: **Use selected text as query**, **Use entire document as query**, and **Determine search scope**. In order to select text to use as a query, the user must be able to highlight it by some obvious point-and-click regimen. This should be the same regimen used for copying text, except that the user selects a **Search** menu option rather than the standard "copy" action.

MENUS:

Menu	Item	Action
File	Save Quit	saveText()
Edit	Cut Copy Paste	Only Copy available.
Search	Use selected text as query Use entire document as query Determine search scope	searchText() searchText() setSearchScope()
Help	On searching On search scope	

CALLBACKS:

```
int
saveText(+Title, +Text)      /* NOTE: This message may be caught and executed by the local user */
char* Title;                /* interface manager, rather than passed to the underlying system. */
char* Text;                 /* The current configuration of the text object. */
```

```
int
searchText(+Title, +SelectedText)  /***NOTE: Only a single callback is needed here, since */
char *Title;                       /* the two actions differ only from the user's point of view */
char *SelectedText;                /* and in the amount of text sent, not in the effect of the */
/* call. */
```

```
int
setSearchScope(+Title)
char *Title;
```

CHILDREN:

Text Object Creation (q.v.)
Text Collection Display (q.v.)
Text Node Display (q.v.)

Appendable Text Display: additional methods:

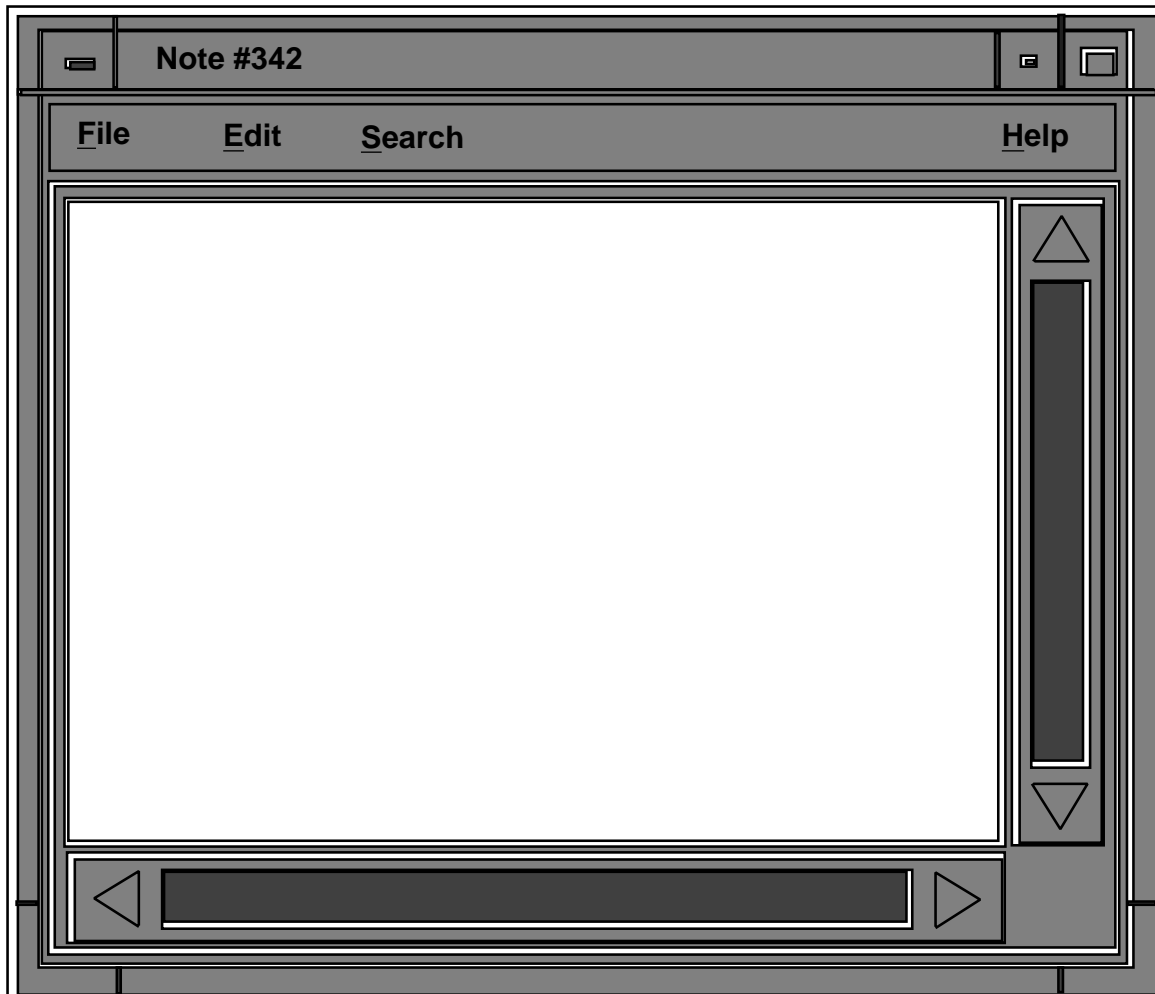
```
int
appendToText(+Title, +Text)      /* Adds Text to the bottom of the text */
char *Title;                     /* currently in the window named Title. */
char *Text;
```

Text Creation Object

38

```
int  
solicitText(+Title, +InitialText)  
char *Title;  
char *InitialText;
```

e.g.: solicitText("Note #342", "")



NOTES:

In addition to **Quit** and **Save**, the **File** menu contains an **Open** option. Again, this is handled by the user interface manager, which conducts a dialog with the user to determine the location of the file to be opened, reads in the text, and passes it back to the window.

The **Edit** menu has all three functions available and active.

This class is really a triplet of three closely related classes. Objects of the basic textCreationWindow class should simply remain unchanged when an action is chosen via the **Search** menu. Two sibling classes can be determined, however, both of which have been used in some CODER systems. The blankingTextCreationWindow (call: solicitTextBlanking()) has its editable field cleared whenever the text is sent to CODER. The vanishingTextQueryWindow (call: solicitTextVanishing()) "goes away" whenever a query is sent. If these variants are difficult to implement separately, note that they can be constructed externally using the goAway() and clear() methods defined in the parent non-modal class.

MENUS:

Menu	Item	Action
File	Open Save Quit	callback: readText()
Edit	Cut Copy Paste	<< All available. >>
Search	Use selected text as query Use entire document as query Determine search scope	<< inherited from textDisplay class >>
Help	On CODER / On MARIAN On searching On search scope	

CALLBACKS:

```

int
readText(-Text)          /* NOTE: This message may be caught and executed by the local user */
char* Text;              /* interface manager, rather than passed to the underlying system. */

int
newText(+Title, +SelectedText)
char *Title;
char *SelectedText;

```

CHILDREN:

Query Creation (q.v.)

SIBLINGS:

Blanking Text Creation: additional methods: none.

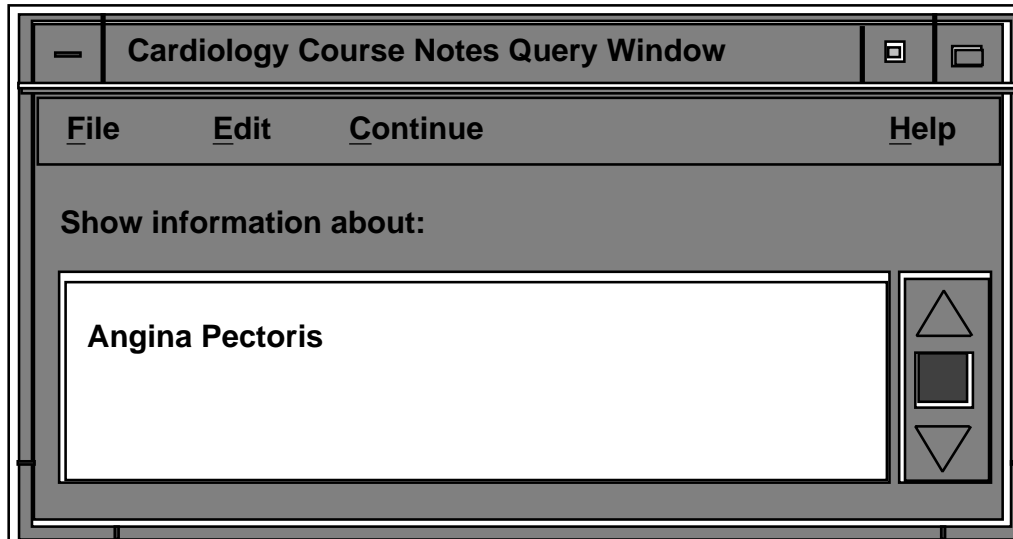
VanishingText Creation: additional methods: none.

Query Creation Object

40

```
int  
solicitQuery(+Title, +Prompt)  
char *Title;  
char *Prompt;
```

e.g.: `solicitQuery("Cardiology Course Notes Query Window", "Show information about:")`



NOTES:

This is a minor variant on the `textCreationWindow` class. The major differences are the size of the editable field and the addition of the Prompt text. The **Search** menu and its items have been renamed, but their effect and the callback that they evoke is the same as in the parent. A vertical scroll bar should appear when the user's text (here, "Angina Pectoris") exceeds the size of the editable field; text should wrap, though, rather than the field expand horizontally.

MENUS:

Inherited from `textCreationWindow` class, with the substitution for Search of:

Continue Search Perform search callback: `searchText()`, as in `textObjDisplay` class

and the addition of:

Help On constructing queries.

CALLBACKS:

Inherited from `textCreationWindow` class.

CHILDREN:

Journal Article Form Query Creation Window (q.v.)
Bibliographic Form Query Creation Window (q.v.)

Journal Article Form Query Creation Object

42

```
int  
solicitJrnArtQuery(+Title)  
char *Title;
```

e.g.: `solicitJrnArtQuery("Medlars journal article database search:")`

The image shows a graphical user interface window titled "Medlars journal article database search:". The window has a menu bar with "File", "Edit", "Continue", and "Help". Below the menu bar are four input fields: "Author(s)", "Words in:", "Journal:", and "Year(s)". The "Words in:" field has three radio buttons labeled "Title", "Abstract", and "All", with "All" selected. To the right of each input field are navigation controls: up and down arrows and a central button.

NOTES:

Except for the trio of radio buttons, this window is basically four textQuery windows pasted together, and all comments on that class and its children apply.

Any or all of the text arguments in the callback may be the null string. The KeyWordCoverage argument encodes the selection from the three radio buttons left of the "Words in:" field: 0=All, 1=Title, 2=Abstract.

MENUS:

Inherited from queryCreation class, with the addition of:

Continue search	Use selected text as query. Use current field as query. Use entire form as query.
Help	On the journal article form.

CALLBACKS:

```
int  
jrnlArtQueryText(+Title, +AuthorStr, +KeyWordCoverage, +KeyWordStr, +SourceStr, +DateStr)  
char *Title;  
char *AuthorStr;  
int KeyWordCoverage;  
char *KeyWordStr;  
char *SourceStr;  
char *DateStr;
```

Bibliographic Form Query Creation Object

44

```
int  
solicitBiblioQuery(+Title, +AuthorStr, +Text1Coverage, +Text1Str, +Text2Coverage, +Text2Str,  
                  +Text3Coverage, +Text3Str, +DateStr)  
char* Title;  
char* AuthorStr;  
int Text1Coverage;  
char* Text1Str;  
int Text2Coverage;  
char* Text2Str;  
int Text3Coverage;  
char* Text3Str;  
char* DateStr;
```

e.g.: solicitBiblioQuery("Query #1:", "", 1, "cat dog", 2, "feline canine", 15, "", "")

[[Drawing on next page.]]

NOTES:

This is an object for gathering up to five text fields: one to be filled with (partial descriptions of) the author(s) of a work, one for the date of the work, and two to three others, each of which can be set to a number of alternatives. The alternatives are still not completely determined at this point, but include at least: "Title", "Subject", "Notes", "Title + Subject", "Title + Author", and "All". Where possible, these should be implemented as a "pop-up menu", with default values "Title", "Subject", and if space permits a third configurable text field, "All". Except for the alternative selection mechanism, this object is basically five textObjQuery windows pasted together, and all comments on that class and its children apply.

Any or all of the text arguments in the callback may be the null string. The Text*Coverage arguments encode the selection from the alternatives, with: 1=Title, 2=Subject, 3=Title+Subject, 4=Notes, 5=Title+Notes, 8=Author, and so forth. Since Title, Subject, Notes and Author are the only atomic values thus far defines, 15 is equivalent to All.

Query #1: □ □

File Edit Continue Help

Author(s): ▲
▬
▼

Words in: ▲
 ▬
▼

Words in: ▲
 ▬
▼

Words in: ▲
 ▬
▼

Year(s): ▲
▬
▼

MENUS:

Inherited from queryCreationWindow class, with the callback changed in:

Continue Search Perform search callback: biblioQueryText()

and the addition of:

Help On the bibliographic query form
 Re-using previous queries

CALLBACKS:

```
int
biblioQueryText(+Title, +AuthorStr, +Text1Coverage, +Text1Str, +Text2Coverage, +Text2Str, +Text3Coverage,
                +Text3Str, +DateStr)
char *Title;
char *AuthorStr;
int Text1Coverage;
char *Text1Str;
int Text2Coverage;
char *Text2Str;
int Text3Coverage;
char *Text3Str;
char *DateStr;
```

Simple Bibliographic Form Query Creation Object

47

```
int  
solicitSimpleBiblioQuery(+Title, +AuthorStr, +Text1Str, +Text2Str, +Text3Str, +DateStr)  
char *Title;
```

```
e.g.: solicitSimpleBiblioQuery("Query #1:", "", "", "", "", "")
```

[[Drawing on next page.]]

NOTES:

This is a simpler version of the preceding window, where neither the system nor the user can change the coverage of the flexible text fields. It is defined for the convenience of implementors in environments with no "pop-up" facilities, but might also be used for naive and confused users.
















The callback is the same as for the full class, with fixed values for the Text*Coverage arguments.

CALLBACKS:

```
int  
biblioQueryText(+AuthorStr, +Text1Coverage, +Text1Str, +Text2Coverage, +Text2Str, +Text3Coverage,  
               +Text3Str, +DateStr)  
char *AuthorStr;  
int   Text1Coverage = 1;  
char *Text1Str;  
int   Text2Coverage = 2;  
char *Text2Str;  
int   Text3Coverage = 0;  
char *Text3Str;  
char *DateStr;
```


Query #1: [] []

File **Edit** **Continue** **Help**

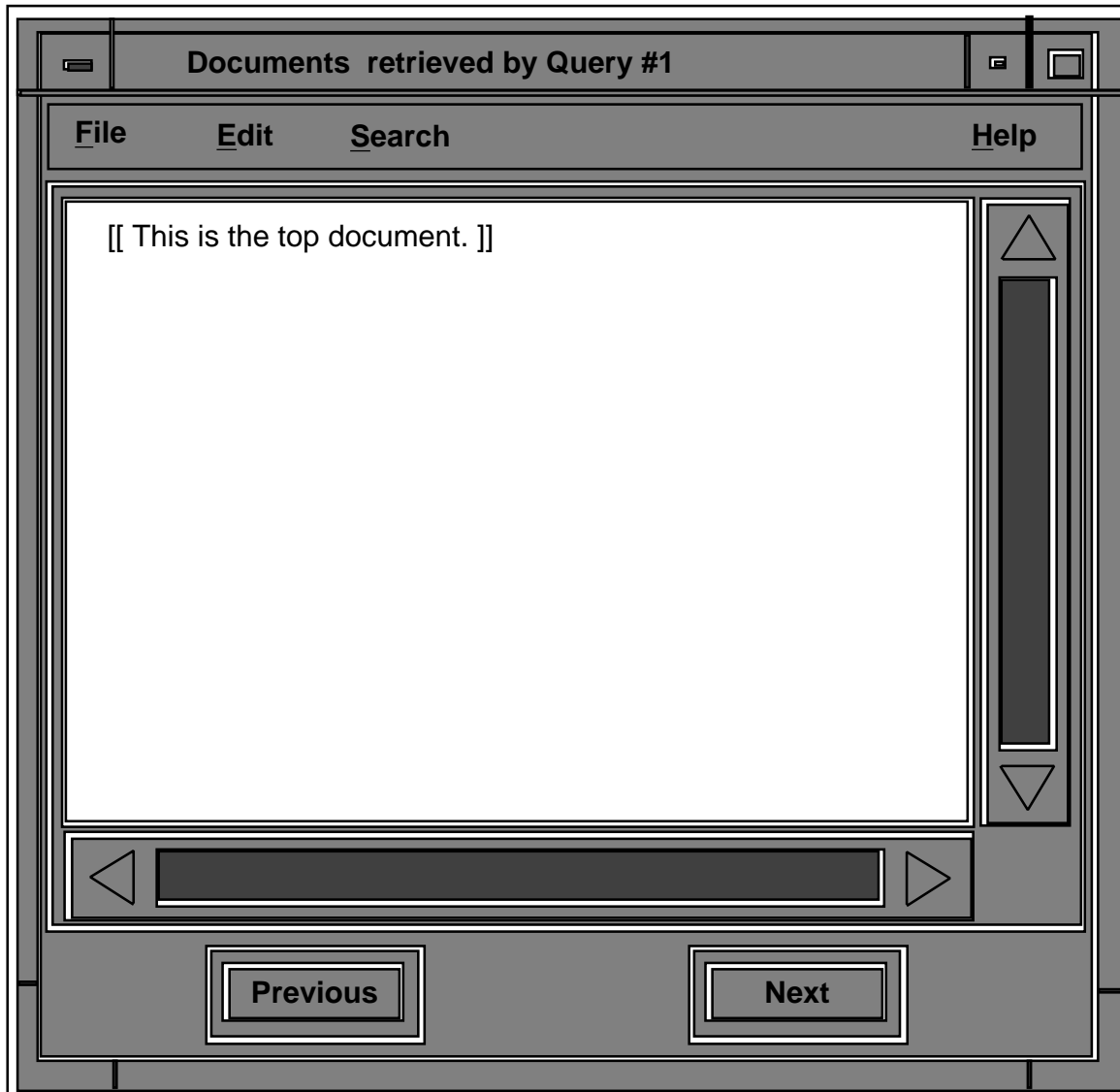
Author(s):	<input type="text"/>	  
Words in Title	<input type="text"/>	  
Words in Subject	<input type="text"/>	  
Words Anywhere	<input type="text"/>	  
Year(s):	<input type="text"/>	  

Text Collection Display Object

49

```
int  
showTextColl(+Title, +FirstText)  
char *Title;  
char *FirstText;
```

e.g.: `showTextColl("Documents retrieved by Query #1", "[[This is the top document.]] ")`



NOTES:

This window is used to display closely bound groups of texts – such as documents retrieved by a single query or pages in a book – in an orderly way. Multiple calls to `addToTextColl()` establish a sequence of texts that the user can page through using the **Previous** and **Next** buttons. What happens at the ends of the sequence are not specified here, but should be some obvious action idiomatic to the interface style, such as wrapping around to the other end of the list, repeatedly displaying the end document, or displaying a message to the user. In any case, the action should be local; no callbacks are associated with the buttons.

MENUS & CALLBACKS:

Inherited from textObjectDisplay class.

ADDITIONAL METHODS:

```
int  
addToTextColl(+Title, +Text)          /* Add to the end of the group of texts */  
char *Title;                          /* currently being displayed; not to the end */  
char *Text;                            /* of any single text. */
```

CHILDREN:

Course Notes Collection Display (q.v.)
Text Node Collection Display (q.v.)

Book Display Object

```
int
showBook(+Title, +FirstText)
char *Title;
char *FirstText;
```

e.g.: showBook("Cardiology Course Notes", "\bClinical Manifestations of Cardiac Ischemia\n\n\uAngina Pectoris\u\b\n\n Angina is a discomfort that results [[etc.]]")

[[Drawing on next page.]]

NOTES:

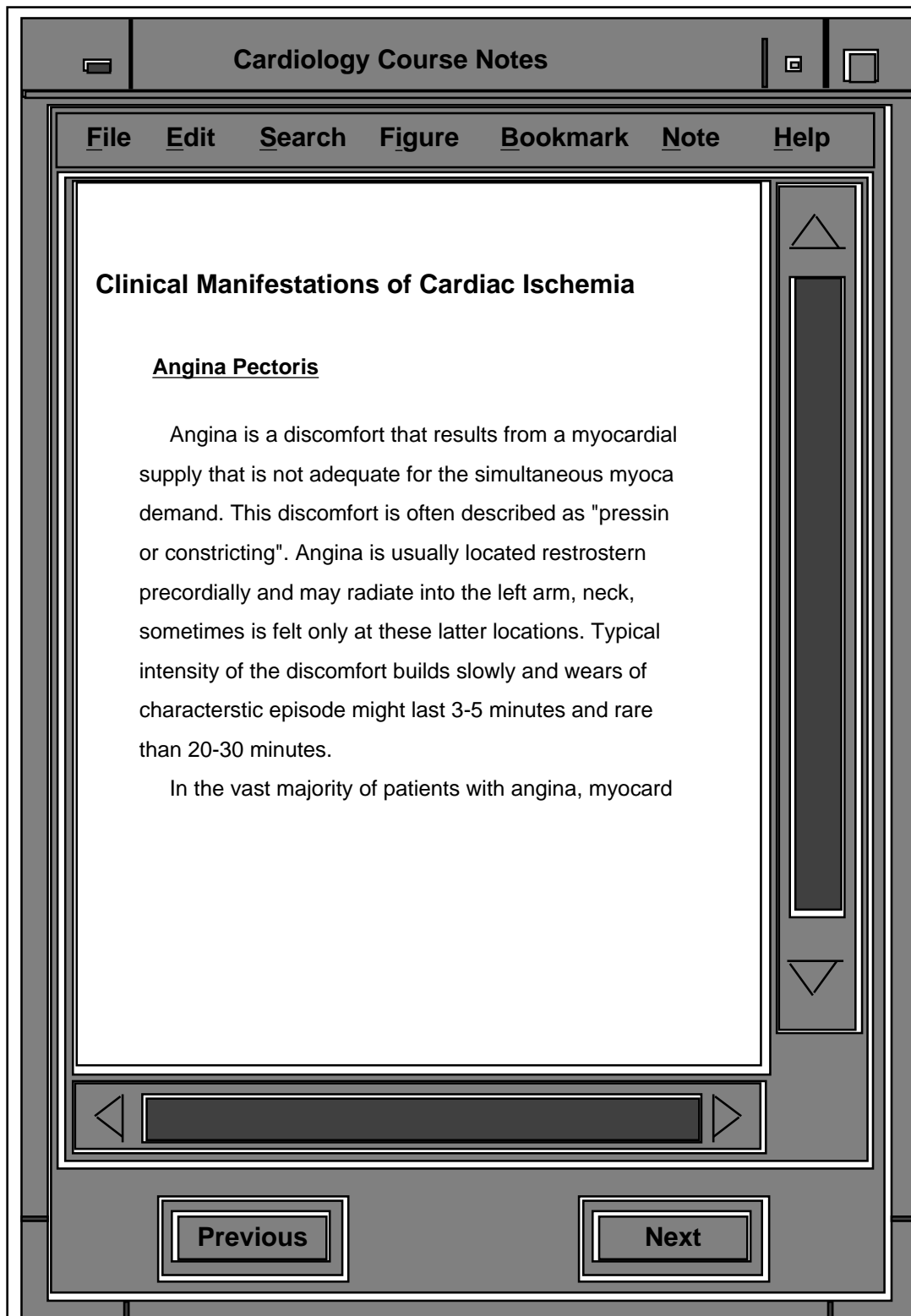
This is a prototype for a hyperbook display window. Besides the normal flowing text with (optionally) font control and style markings, the text also contains specialized markers for figures, footnotes, and bookmarks. What these markers are is unimportant to the window implementation, as the means for following one of them to the associated figure, note, or place in text is the same as that for editing or using part of the text as a query: the special marker is selected, and a choice made from the **Figure**, **Note**, or **Bookmark** menu. This choice generates a callback that results in a new text being added to the collection or a new figure displayed in a graphics window (not covered in this document).

Creating new bookmarks and notes, in contrast, does require local processing. When the user selects **Create note** from the **Note** menu or **Place bookmark** from the **Bookmark** it should first cause a newNote() or newBookmark() callback, which alerts the body of CODER to the change and results in a string for the new marker. That string then needs to be inserted by the window manager at the cursor position, and any resulting updates handled.

The exact means of updating document text through addition of notes and bookmarks is still under discussion. Two possibilities dominate: either changes are made incrementally with each call to newNote(), newBookmark(), deleteNote() or deleteBookmark(); or all changes are accumulated until the user chooses a **Commit** option from the **File** menu. This difference creates differences in the callbacks: in the first case, the position of the insertion or deletion within the text must be passed to CODER proper, so that changes can be made in the master copy of the text as well as in the local window. This may be difficult for the window manager, as it requires translating cursor position into a byte offset from the beginning of the text. In the second, the entire text can be shipped, with changes intact, upon selection of the **Commit** option, so the problem does not arise. But this latter option requires sending large texts back to CODER, with an associated transmission cost. In either case, some provision must be made for distinguishing which text among the collection that the user is paging through is currently being altered. The means for even this have yet to be established. The callbacks noted here are for the second option.

ADDITIONAL METHODS:

```
int
addToBook(+Title, +Text)
char *Title;
char *Text;
```



MENUS:

Inherited from textDisplay class, with the additions:

Menu	Item	Action
[[File	Commit	updateDoc()]]
Figure	Show selected figure Show all figures	textLink()
Bookmark	Show/hide all bookmarks Place bookmark Remove bookmark Go to bookmark	newBookmark() deleteBookmark() textLink()
Note	Show/hide notes Create note Delete note Open note	newNote() deleteNote() textLink()
Help	On figures On bookmarks On notes	

CALLBACKS:

Inherited from textDisplay class, with the additions:

```

int
textLink(+Title, +LinkText)
char *Title;
char *LinkText;          /* The selected text of an in-document link: either a figure, note, or */
                          /* bookmark marker. The "all figures" option is accomplished using */
                          /* multiple calls to this callback. */

int
newNote(+Title, -NoteMarker)
char *Title;
char *NoteMarker;       /* The text of the marker to be entered into the document displayed. */

int
deleteNote(+Title, +NoteMarker)
char *Title;
char *NoteMarker;       /* The text of the marker to be deleted. */

int
newBookmark(+Title, -BookmarkMarker)
char *Title;
char *BookmarkMarker;   /* The text of the marker to be entered into the document displayed. */

int
deleteBookmark(+Title, +BookmarkMarker)
char *Title;
char *BookmarkMarker;   /* The text of the marker to be deleted. */

int
updateDoc(+Title, +Text)
char *Title;
char *Text;

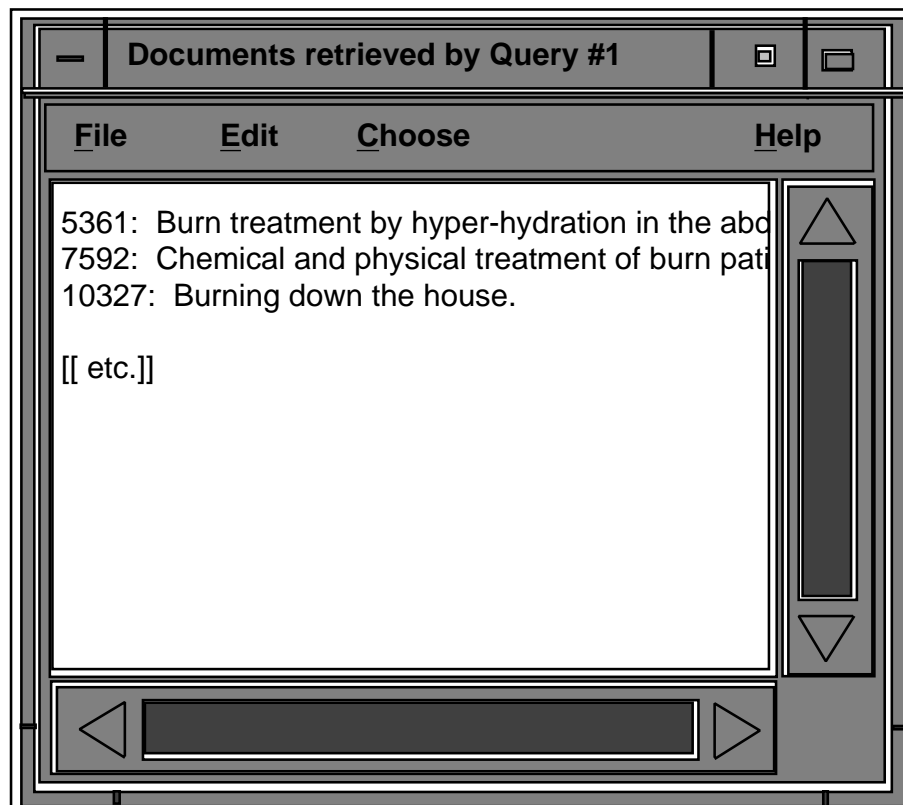
```

Choice List Display Object

55

```
int  
showChoiceList(+Title, +ChoiceVect , +NumChoices, +SuggestedChoiceNum)  
char *Title;  
char *ChoiceVect[];  
int NumChoices;  
int SelectedChoiceNum; /* 0="No selection", n<0 = "Center and highlight choice #n". */
```

e.g.: showChoiceList("Documents retrieved by Query #1",
{ "5361: Burn treatment by hyper-hydration in the abdominally injured rat.", "7592:
Chemical and physical treatment of burn patients.", "10327: Burning down the
house.", [[etc.]] }, 25, 0)



NOTES:

The **Choose** menu has two choices: **Choose selected item(s)** and **Choose all**. Items should be selectable by pointing anywhere on the item and mousing. If it fits the X idiom, it should be possible to choose an item by simply pointing and clicking a different button. Pointing and dragging, or following some other idiomatic way of selecting multiple objects, result in several choices being selected and sent back to CODER.

The **Edit** menu has only the **Copy** option available. Since the smallest selectable piece of text is a single line, the text copied should always consist of a single line.

ADDITIONAL METHODS:

```

int
addToChoiceList(+Title, +ChoiceVect , +NumChoices)
char *Title;
char *ChoiceVect[];
int NumChoices;

```

MENUS:

Menu	Item	Action
Choose	Choose selected item(s) Choose all	userChoice() or userChoices() userChoices()
Help	On choosing	

CALLBACKS:

```

int
userChoice(+Title, +Choice)
char *Title;
char *Choice;

```

```

int
userChoices(+Title, +ChoiceVect , +NumChoices)
char *Title;
char *ChoiceVect[];
int NumChoices;

```

CHILDREN:

Text Node Display (q.v.)

Retrieval Set Display (q.v.)

Single Choice List Display: exactly like its parent, except that only a single item from the list of choices may be selected or chosen at a time.

Text Node Display Object

```

int
showTextNode(+Title, +Text, +ChoiceVect , +NumChoices, +SelectedChoiceNum)
char *Title;
char *Text;
char *ChoiceVect[];
int NumChoices;
int SelectedChoiceNum;      /* 0="No selection", n<0 = "Center and highlight choice #n".  */

```

```

e.g.: showTextNode("Amyl Nitrate", "A colorless, odorless gas used in treating various cardiac conditions:
the nitrate of a univalent hydrocarbon radical C5H11 derived from pentane.",
{"MeSH Descriptors", " Angina Pectoris", "Medlars documents", " 63824", " 23892",
" 98153", " 32411", " 2341", " 54245"}, 9, 0)

```

[[Drawing on next page.]]

NOTES:

This is basically a textObjectDisplay and a choiceListDisplay object pasted together. The menu items and callbacks are determined by the parents, with the **C**hoose menu being renamed **L**ink.

CHILDREN:

Text Node Collection Display (q.v.)

The image shows a screenshot of a web browser window. The title bar at the top reads "Amyl Nitrate". Below the title bar is a menu bar with the following items: "File", "Edit", "Search", "Link", and "Help".

The main content area is divided into two sections:

Text.

A colorless, odorless gas used in treating various cardiac conditions: the nitrate of a univalent hydrocarbon radical C₅H₁₁ derived from pentane.

Links connected to this text.

- MeSH Descriptors
 - Angina Pectoris
- Medlars documents
 - 63824
 - 23892
 - 98153
 - 32411

The browser interface includes a vertical scrollbar on the right side of each text area and a horizontal scrollbar at the bottom of each section.

Text Node Collection Display Object

59

```
int
showTextNodeColl(+Title, +FirstText, +FirstChoiceVect , +NumChoices, +SelectedChoiceNum)
char *Title;
char *FirstText;
char *FirstChoiceVect[];
int NumChoices;
int SelectedChoiceNum;      /* 0="No selection", n<0 = "Center and highlight choice #n". */
```

e.g.: showTextNodeColl("wolf [1,1,1]", "any of a class of animals of the genus lupus, including the gray wolf and the European timber wolf.", {"Full Entry", "Compare (forward)", " timber wolf [1]", "Related Adjectives", " lupine [1]"}, 5, 0)

[[Drawing on next page.]]

ADDITIONAL METHODS:

```
int
addToTextNodeColl(+Title, +Text, +ChoiceVect , +NumChoices)
char *Title;
char *Text;
char *ChoiceVect[];
int NumChoices;
```



MeSH Node Display Object

61

```
int
showMeshNodeColl(+Title, +FirstText, +FirstHierVect , +NumHiers, SelectedHierNum, +FirstEntryVect,
                 +NumEntries, SelectedEntryNum, +FirstMinorDescVect , +NumMinorDescs,
                 +SelectedMinorDescNum, +FirstMajorDescVect, +NumMajorDescs,
                 +SelectedMajorDescNum)

char *Title;
char *FirstText;
char *FirstHierVect[];
int NumHiers;
int SelectedHierNum;
char *FirstEntryVect[];
int NumEntries;
int SelectedEntryNum;
char *FirstMinorDescVect[];
int NumMinorDescs;
int SelectedMinorDescNum;
char *FirstMajorDescVect[];
int NumMajorDescs;
int SelectedMajorDescNum;
```

e.g.: showMeshNodeColl("Descriptor Information", "\bAngina Pectoris\n\n C14.280.211.198+\b\n\nThe symptom ofparoxysmal pain ischemia usually of distinctive ch radiation, and provoked by a tran duringwhich the oxygen required ...", { ..., "Coronary Disease C1", " Angina Pectoris C1", " Angina Pectoris, variant C1", " Angina , Unstable C1", " Coronary Aneurysm C1", " Coronary Arteriosclerosis C1", " Coronary Thromobosis C1", ... }, 15, 8, {"Stenocardia", "Angor Pectoris"}, 2, 0, {"Angina, Unstable"}, 1, 0, {"Chest Pain"}, 1, 0)

[[Drawing on next page.]]

NOTES:

A texttDisplay object and four choiceListDisplay objects pasted together. The **Link** menu still invokes the same userChoice() and userChoices() callbacks as used in the case with only a single class of links, as the type of link followed to a destination is immaterial to the action of displaying the destination.

The **Link** menu does include an additional user action, "To MEDLARS collection", used to display documents in MEDLARS indexed by the current MeSH term.

Descriptor Information

File

Edit

Search

Link

Help

Term Description

Angina Pectoris

C14.280.211.198+

The symptom of paroxysmal pain ischemia usually of distinctive ch radiation, and provoked by a tran during which the oxygen required

Location in Hierarchy

Coronary Disease	C1
Angina Pectoris	C1
Angina Pectoris, variant	C1
Angina , Unstable	C1
Coronary Aneurysm	C1
Coronary Arteriosclerosis	C1
Coronary Thromobosis	C1

Entry Terms referring to this descriptor

Stenocardia
Angor Pectoris

Minor descriptors referring to this descriptor

Angina, Unstable

Major descriptors referring to this descriptor

Chest Pain

Previous

Next

MENUS & CALLBACKS:

Inherited from textNodeCollDisplay class, with the additions:

Menu	Item	Action
Link	To MEDLARS colection	linkMeshToMedlars()
Help	On MeSH	

ADDITIONAL METHODS:

```

int
addToTextNodeColl(+Title, +Text, +HierVect, +NumHiers, +EntryVect, +NumEntries,
                  +MinorDescVect, +NumMinorDescs, +MajorDescVect, +NumMajorDescs)
char *Title;
char *Text;
char *HierVect[];
int NumHiers;
char *tEntryVect[];
int NumEntries;
char *MinorDescVect[];
int NumMinorDescs;
char *MajorDescVect[];
int NumMajorDescs;

```

CALLBACKS:

```

int
linkMeshToMedlars(+Title, +Text)
char *Title;
char *Text;

```


Retrieval Set Display Object

```

int
showRetrievalColl(+Title, +RetrVect, +NumRetr, +MoreToCome)
char *Title;
struct
{
    F3IFullIDType ID;
    char *ShortDesc;
    char *FullDesc;
} RetrVect[];
int NumRetr;
Bool MoreToCome;

```

```

e.g.: showRetrievalColl("Documents retrieved by Query #1",
    {{#119:235134#, "Bridge, Jane: Beginning model theory", {...}},
    {#119:907871#, "Barwise, John (ed): Model theoretic logics", {...}},
    {#119:907871#, "Chang & Keisler: Model theory", {...}}, 20, TRUE)

```

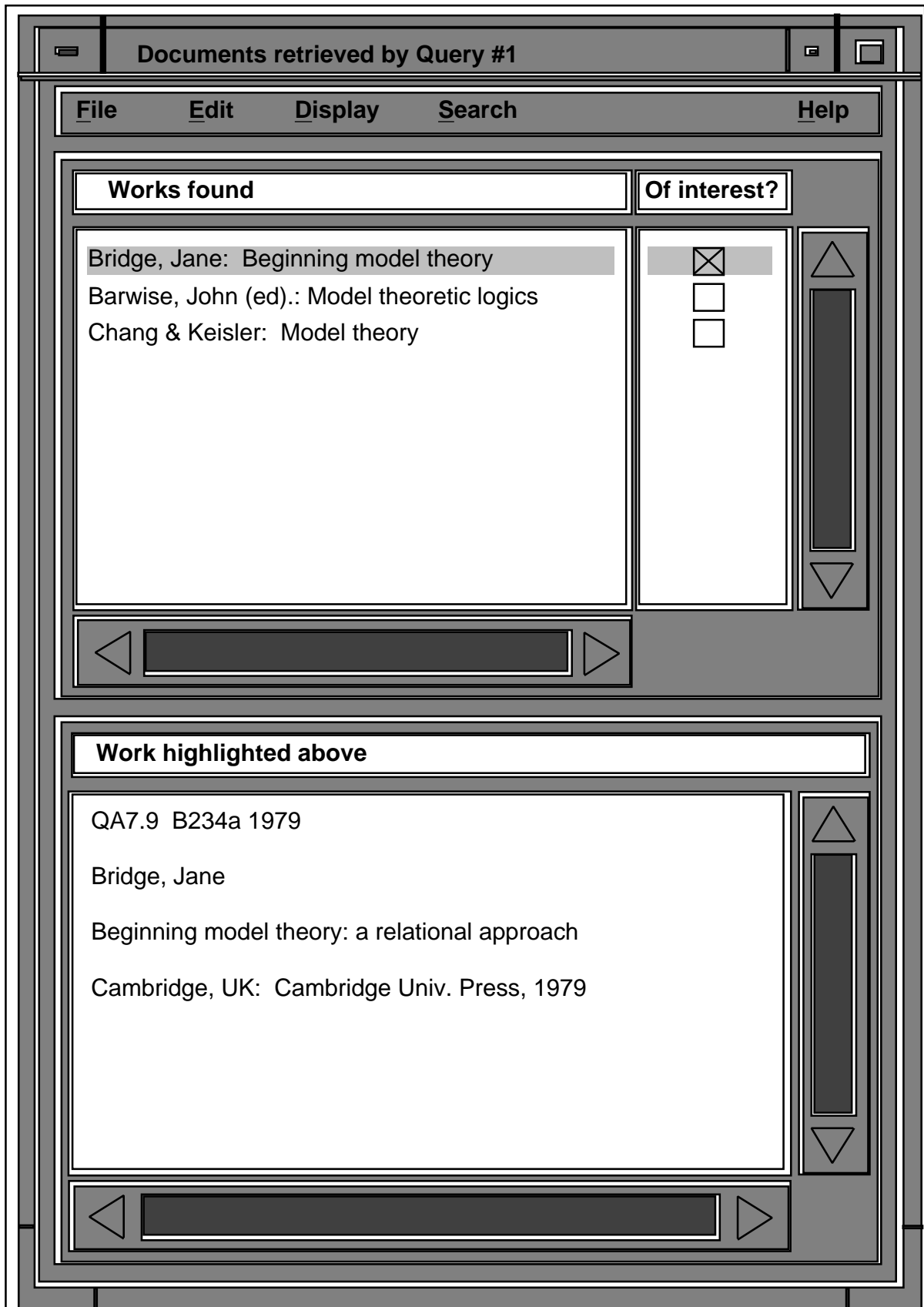
[[Drawing on next page.]]

NOTES:

This is another composite of a textDisplayObject and a choiceDisplayObject, but in this case the texts corresponding to each choice are input with the choice list. Consequently, there is no callback associated with selecting a choice; instead, the object works locally to change the text.

The list of checkboxes labelled "Of interest?" is meant to scroll together with the choice list under the command of the single vertical scroll bar in the upper panel. The checkboxes are initially all in the "off" state; the user points and clicks to set them "on". When the "From documents of interest" option is selected from the **Search** menu, the IDs associated with the "on" checkboxes are returned; if no boxes have been set to "on", a modal information window should be spawned locally to instruct the user.

If MoreToCome is set to TRUE, then the "Find more using this query" option should be enabled when the object is created. This allows the user to get a continuation of the retrieval set by initiating a showNextK() callback, which in turn triggers an addToRetrievalColl() call from the system. The status of the option is then determined by the MoreToCome argument of the addToRetrievalColl() call, and so forth until a FALSE is received, at which time the option is disabled.



MENUS:

Menu	Item	Action
Continue Search	From current document	feedbackSearch()
	From documents of interest	"
	Find more using this query	showNextK()
Help	On display options	
	On feedback searches	

CALLBACKS:

```
int
feedbackSearch(+Title, +DocIDVect, +NumDocs)
char *Title;
F3IFullIDType DocIDVect[];
int NumDocs;
```

```
int
showNextK(+Title, +K)
char *Title;
int K;
```

ADDITIONAL METHODS:

```
int
addToRetrievalColl(+Title, +RetrVect, +NumRetr, +MoreToCome)
char *Title;
struct
{
    F3IFullIDType ID;
    char *ShortDesc;
    char *FullDesc;
} RetrVect[];
int NumRetr;
Bool MoreToCome;
```