BROWN UNIVERSITY
Department of Computer Science
Master's Project

CS-94-M15

"A System for Supporting Mark-based Interaction in Motif"

by

Lalit K. Agarwal

A System for Supporting Mark-based Interaction in Motif

Lalit K. Agarwal
Department of Computer Science
Brown University
Providence, RI 02912

A System for Supporting Mark-based Interaction in Motif

Lalit K. Agarwal
Department of Computer Science
Brown University
Providence, RI 02912

Submitted in partial fulfillment of the requirements for the degree of Master of Science in the Department of Computer Science at Brown University.

May 1994

Professor Steve Reiss

Advisor

Supporting Mark-based Interaction In Motif

Lalit K. Agarwal
Dept. of Computer Science
Brown University, Providence, RI 02912

1.0 Abstract

A system is presented to augment Motif widgets with mark-based interaction without affecting their normal functionality, and without having to modify any internal Motif code. A model of interaction is provided in which an interface object is placed between a widget and the application to handle marking interaction. The interface processes a mark made in a widget, classifies it, and upon successful classification, calls a pre-specified call-back function in the application. The call-back function is passed a data structure that contains pertinent information about the classified mark. The interface also allows one to view the marks recognized by it, to activate (deactivate) particular marks in a widget, and to activate (deactivate) marking interaction in a widget, using simple key (+button) combinations. An interface object classifies a mark with the help of a classifier embedded in it. An application is provided to create, store and maintain the classifiers needed for an interface. A classifier is created by providing and training it with examples of the various mark types (or classes) it is intended to classify. In order to facilitate creation of classifiers by using marks from pre-created inventories (or catalogs), the application also allows the creation, storage and maintenance of mark catalogs.

2.0 Introduction

Point-and-Click or direct manipulation interfaces -- like Macintosh, Motif, Windows etc. -- have gained popular and wide acceptance among the computing community during the last few years. They provide an intuitive way of interacting with an application in which the user can directly manipulate graphical representations of the underlying data on the screen with a mouse or pen. By providing graphic objects and forms which resemble objects from daily life, they tend to ease interaction with the computer.

The direct manipulation interfaces, however, suffer from the principal limitation that they provide only a very primitive repertoire of interaction techniques, i.e. point, click, drag etc. As a consequence, the user must express all operations in his (her) application, in terms of these primitive operations. Application designers have chosen primarily two design alternatives to provide more than the primitive number of operations on any object in an application. The alternatives commonly chosen are: "selection-based click-and-drag" approach and "Mode-based" approach. In the first approach, the object to be operated upon is first selected, and then the desired operation is performed by selecting it from a menu. In the second approach, to do an operation on an object, the user must explicitly set an appropriate mode first. For example, in the Macintosh drawing program

MacDraw, to move an object, one first sets the mode to "move" mode and then points and drags the object to the desired new location. Mistakes occur when the user believes he is one mode but is actually in another. Moreover, in both the approaches, the users must adjust their mental model so that they think in terms of the component operations.

Mark-based interaction, a new way of interacting with computers in which the user specifies commands by simple freehand drawings (using the mouse or pen) called marks, offers an alternative to the traditional keyboard and direct manipulation interfaces. This form of interaction finds appeal among both novice and experienced users because of the ability to specify an operation, objects (target of the operation), and additional parameters with a single intuitive mark in it. The intent of mark-based interfaces is to avoid the division of an application command into a sequence of primitive interactions, thereby, simplifying the user's mental model of interaction. Ideally, each basic task in the application is executed with a single mark which packs the basic interaction with all the parameters necessary to complete the entire transaction. Such interfaces would have less modeness than direct manipulation interfaces.

A principal drawback of mark-based interfaces is that the users may have to learn a considerable number of marks. This may turn out to be counter productive instead if the number is large and different sets of marks are used in different parts of the application. Moreover, some of an application's tasks may be best carried out using point-and-click style of interaction, and for which, using a mark might be an overkill. A reasonable alternative interface model therefore seems, should be able to provide both the models of interaction -- namely, direct-manipulation and mark-based. The mapping of an application command to an interaction technique has to be determined carefully (keeping the usability of the interface in mind) by the application designer. Striking the right balance between the two interaction techniques in an application would not be a trivial task, but, once done right, may greatly improve the usability of the application. In order to test the viability of such an interface, it seems a sound strategy at this point to incorporate the marking model of interaction in an existing (and obviously, widely accepted) direct-manipulation interface, rather than developing a new interface from scratch. For the purposes of this work, Motif was chosen as the interface to be augmented with marking capabilities, mainly because of its wide acceptance and ready availability of documentation on it and X.

This document describes a technique and a software system for supporting mark-based (or gesture-based) interaction in Motif. The system aims to provide means to augment Motif to support mark-based interaction without affecting any of its normal functionality. A Mark in this system is defined as a free-hand drawing made by using the mouse. A specified mouse button (+ key) needs to pressed while making the mark. The system is divided into two distinct parts: one dealing with the specification of marks and creation of *classifiers* (also called recognizers), and the other dealing with the actual augmentation of the Motif widgets with marking capabilities. These two parts are described in detail in the rest of this document. The source code for this package can be found in Appendix A.

3.0 Specification of Marks and Creation of Classifiers

Before a widget can classify marks, one has to first specify to it which marks to classify, and then, this information (perhaps, after some pre-processing) has to be linked to the widget. The specification part of the system deals with this important aspect. Mark-based systems require classifiers to distinguish among the possible marks the user may make. A classifier takes the mark made by the user, and tries to classify it. The classifier thus provides the underlying intelligence to support mark-based interaction. Therefore, for every widget in which mark-based interaction is intended, a classifier would be needed to classify the marks made in that widget. Statistical pattern recognition techniques are used to produce mark classifiers that are trained by example, thereby greatly simplifying their creation and maintenance. To create a classifier, one simply needs to input a few examples of each mark class (or mark type). The classifier is then created by training it on the given examples. To make reuse of a past effort and avoid having to input all the examples each time one wishes to create a classifier, it would be convenient to have an inventory (or catalog) of marks around. One could then simply pick the examples of a mark class from a catalog, instead of having to draw them one by one. A package, designed specifically for the above two purposes, called **`MarkEditor`** is provided which facilitates quick creation of classifiers either by using marks from a catalog or making new ones, and, allows creation and maintenance of mark catalogs.

3.1 Classification Technology

Statistical pattern recognition techniques as described in [11] are used to create classifiers.

3.2 MarkEditor

`MarkEditor` is a stand-alone application intended to facilitate quick creation and maintenance of classifiers -- to be subsequently used to augment Motif widgets with mark-based interaction -- and creation and maintenance of mark catalogs. Following is a list of functions supported by `MarkEditor`:

1. Creation, storage and maintenance of classifiers. Creation is supported in three ways:
   - Selection of a set of classes from a catalogue followed by creation of a classifier to classify these classes.
   - Entering new examples of classes (by drawing them using mouse) and creating a classifier using these examples.
   - Mixing the above two approaches in creating a classifier.
2. Browsing the mark classes in an existing classifier.
3. Addition of new classes to an existing classifier, and new examples to existing classes. Also supported is the deletion of classes from a classifier.
4. Creation, Storage and maintenance of mark catalogues. The editor allows browsing of examples of mark classes in a catalog, addition of new examples to an existing class, addition of new classes to a catalog, and deletion of classes from a catalogue.

Both a catalog and a classifier support a concept of *representative example* of a mark class. A representative example of a class (as the term means) provides a representative shape of mark belonging to that class. A representative example has to be specified by the creator of the catalog (classifier) and is not automatically generated by `MarkEditor`. Once, specified, it is stored as part of a catalog (classifier) and can be viewed (and changed) at the will of the user.

3.3 User Interface of MarkEditor

The user interface of `MarkEditor` consists of 4 main parts or areas: *Menubar*, A pair of *Class-name Lists, Drawing Area* and a *Message Area*. One can open or create multiple catalogs and classifiers at the same time in the editor (`MarkEditor`). However, the editor allows only one catalog and one classifier to be accessible (for browsing, editing purposes) to the user at any given time. The editor achieves this by maintaining a currently active (or selected) catalog and classifier at any time. The various commands available under the menus are divided into two sets: one set valid for the active catalog, and the other valid for the active classifier. The four main constituents of the user interface are described below:

1. **Menubar:** It Comprises "File" and "Edit" menus. Both these menus are divided into two parts: one for catalogs and the other for classifiers. This is self evident once one opens the menus. There is an option (one each for classifiers and the catalogs) under the "Edit" menu for selecting a catalog (classifier) from among the currently opened ones. This provides a means of changing the currently active classifier (catalog) and switching among different catalogs (classifiers).

2. **Class-name Lists:** Two lists, present on the left of the UI, show the list of class names of marks contained in the currently selected catalog and classifier, respectively. The lists will be henceforth called: catalog list and classifier list. Each list displays the name of the currently selected file on its top.

   By clicking on a class name in a list, one can look at the representative example of that class. In order to select multiple items from a list, one needs to keep the `<Control>` key pressed while clicking on the desired items. Every selected item is highlighted to confirm its selection. To validate a multiple selection, one needs to press the `<Return>` key after completing the selection. The lists are automatically updated every time one selects or opens a new catalog (classifier), or adds (deletes) item from a catalog (classifier).

3. **Drawing Area:** This occupies the central region of the UI. This is where marks are drawn (by the user) or displayed.

4. **Message Area:** This occupies the bottom portion of the UI. Any message from the editor is displayed in this area. The messages are numbered and can be scrolled using the scroll bar present on its bottom right.

3.3.1 Catalog Commands

All the commands pertaining to a catalog are placed under the "Catalog" heading in the "File" and "Edit" menus. Following are the operations supported on a catalog:

- **Creating a new catalog:** To create a new catalog, use the "new" option in the "File" menu.
- **Adding marks to a catalog** *(new or old class):* Use the "Add Mark Examples" option under the "Catalog" heading in the "Edit" menu. This command first looks for any currently selected class name in the catalog list. A class name is selected from the list simply by clicking on the name with the left mouse button. If no name is selected, the editor asks for a class name. If the name provided is that of a new class, the editor confirms that with the user before adding the new class to the catalog. After all this, it displays a message in the message area prompting one to draw example(s) of that class in the drawing area. Once done inputting examples, one can indicate so by pressing the key <d> in the drawing area.
- **Removing a Class:** To remove a class from a catalog, first select the class name from the catalog list. Then use the "Remove Class" option under the "Edit" menu. This removes all the examples (along with its representative example) of the class from the catalog and also deletes its name from the catalog list. The lose is permanent and irreversible.
- **Setting Representative Example of a Class:** Use the "Set Rep. Example" option under the "Edit" menu. One can set the representative example of only an existing class.
- **Opening an Existing Catalog:** Use the "Open" option under the "File" menu. One can have multiple catalogs (the limit is 10) open at a time. However, If one exceeds the limit, and wants to open another catalog, one has to first close an existing one. The opened catalog is automatically made the active one.
- **Closing a Catalog:** Use the "Close" option under the "File" menu. It brings up a list of all the currently opened catalogs. The names of those that have been modified since opened or last saved, are marked with an asterix "*". Closing such catalogs would lose any changes made to them since the their last save (if any).
- **Save a Catalog:** Use the "Save" option under the "File" menu to save the currently selected (active) catalog. One can also save all the open catalogs using the "Save All" option. Moreover, one can save the currently selected catalog under a different name using the "Save As" option under the same menu.
- **Viewing the Examples of a Class:** Use the "Show Examples" option in the "Edit" menu.
- **Viewing the Representative Examples of a Class:** Simply click on the class name in the catalog list. The representative example, if any present, will be displayed in the drawing area.

3.3.2 Classifier Commands

All the classifier commands are placed under "Classifier" heading in the "File" and "Edit" menus.

- **Creating a New Classifier:** Use "New" option under the "File" menu.
- **Adding Marks to a Classifier** *(from a catalog)*: Select the classifier and the catalog first. Then, press the "Add Mark(s) from Catalog" option in the "Edit" menu. After that, select marks from the catalog list. To select more than one mark, keep the <Control> key pressed while clicking on the class names in the list. Clicking on a selected name with the <Control> key pressed, unselects that name. After selecting the class names to be added to the classifier, press the <Return> key to commit the selection.
- **Adding Marks to a Classifier** (*by drawing*): One can add new classes and also populate old classes with new examples directly by drawing them in the drawing area. This provides an alternative means of creating a classifier without having a catalog around. Use the "Add Marks (draw)" option in the "Edit" menu.

  There are a few facts one needs to be aware of while inputting examples to train a classifier.

  - The performance of the statistical mark classifier depends on a number of factors. Chief among these are the number of classes to be discriminated between, and the number of training examples per class. As determined by Rubine [11], using 15 examples per class gives good results (99% correct recognition rate with 10 classes, 98% with 15 classes, and 96% with 30 classes).
  - The training examples should have sufficient variability in the features that are irrelevant to the classification of that class. For example, if one intends the classifier to classify a line segment drawn with any arbitrary orientation as a line, then, the examples during training should be entered with wide variations in the orientation. The same argument goes for size of marks.
- **Removing a Mark Class from a Classifier:** First select the class to be removed from the classifier list. Then, press the "Remove Class" option in the "Edit" menu. The class and its examples are lost forever once removed, so one may want to be careful before doing so.
- **Setting Representative Example of a Class:** One can set the representative example of any class in a classifier by using the "Set Rep. Example" option in the "Edit" menu.
- **Classify Examples:** In order to test or play with a classifier, press the "Classify Examples" option in the "Edit" menu. After that, one can draw the marks in the drawing area (using the left mouse button), and the result of the classification is displayed in the message area. Press <d> when done.
- **Open, Save, Close, and Select a Classifier:** Procedures are the same as those above for a catalog.

4.0 Augmenting Motif Widgets with Mark-based Interaction

A model, as shown in Figure 1, is provided for augmenting Motif widgets to support mark-based interaction. In this model, a special interface object, called **`MarkInterface`**, is placed between a widget and the application. The interface embeds the classifier and handles the complete business of mark recognition for the widget. To empower a widget in her application with mark-based interaction capability, the user instantiates a `MarkInterface` object and links it to the widget. This link is established simply by passing a pointer to the widget to the interface,. The link between the `MarkInterface` object and the application is established by passing the interface object the name of an application function which needs to be called upon recognition of a mark. This is similar to the mechanism of call-back functions in Motif. The `MarkInterface` object then does some initial set-up so that all the marking related mouse events are sent to it from the widget. When a mark is made in the widget, the interface first processes it and then with the help of the classifier embedded in it, tries to classify it. Upon recognition, it calls the pre-specified application function, passing along with the information regarding the mark in a data structure.

The concept of the `MarkInterface` object thus provides a powerful yet simple mechanism to extend Motif widgets to handle mark-based interaction. It does not require any changes to the internals of existing widgets.

4.1 MarkInterface

`MarkInterface` provides the interface between a widget and the application for marking- based interaction. Any Motif widget can be empowered to handle mark-based interaction simply by linking it with a `MarkInterface` object. To establish this link, the developer instantiates the interface object and provides it the following set-up information:

1. A pointer to the widget.
2. A pointer to the classifier to be used for the widget, or the name of the file in which the classifier is stored.

4.1.1 Call-back Mechanism

The user can register a call-back function with a `MarkInterface` object using the method:

`MarkSetCallbackProc``(MarkCallbackProc *func, void *userData)`

of `MarkInterface` class. The first argument to this method is the pointer to the application call-back procedure, the optional second argument is any user data that the user wants to be supplied back when the call-back function is called upon recognition of a mark.

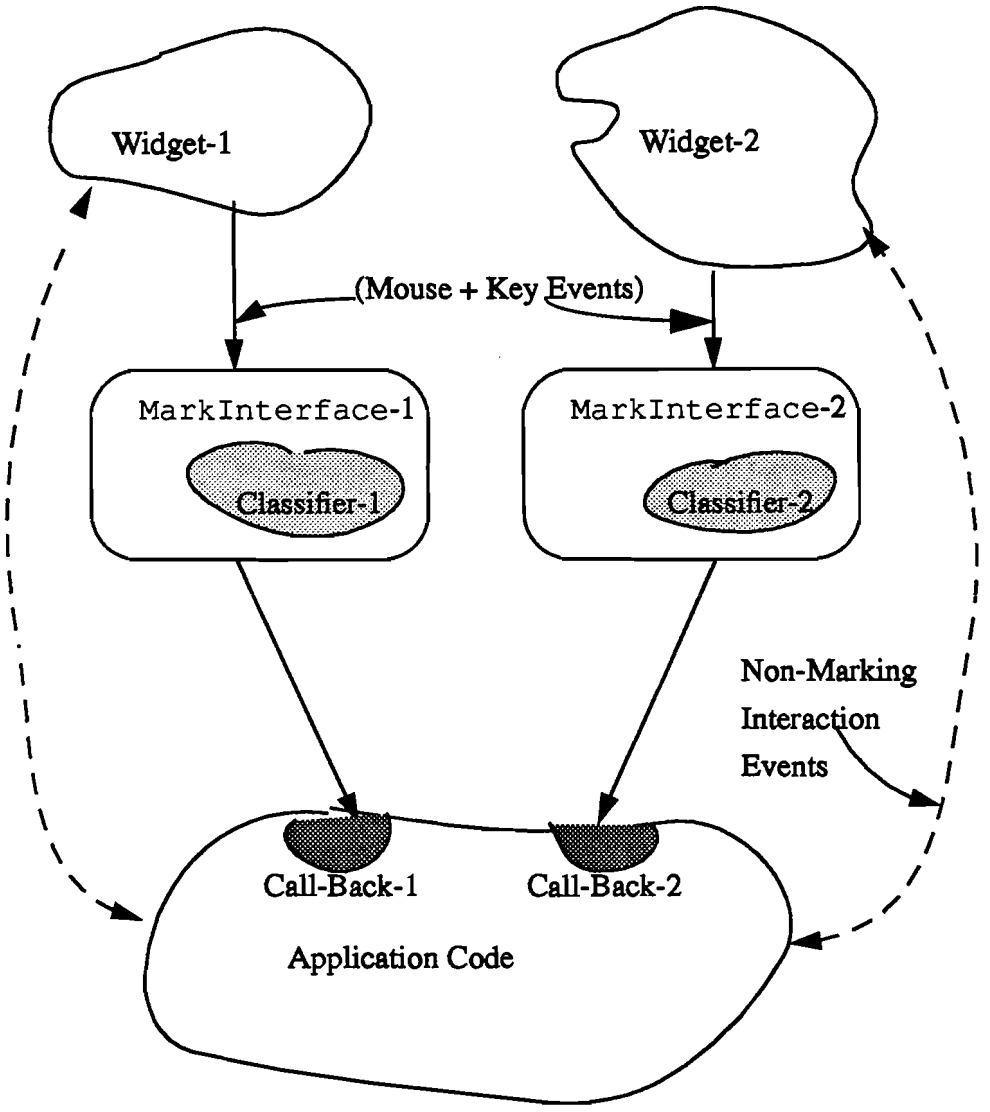

Fig1: Model for supporting Mark-based Interaction in Motif

After creation, a `MarkInterface` object modifies the translation table of the widget so that it can trap all the marking related mouse and key events from the widget.

A `MarkInterface` object:

- Traps all the marking related mouse plus key events from the widget from the start until the end of the mark.
- Feeds the coordinates of the path of a mark to its classifier for a possible classification.
- Upon successful recognition of the mark by the classifier, calls the appropriate call-back function in the application passing it relevant information about the mark. This information about the mark is packaged in a data structure called **`MarkEvent`**.
- Provides means to replace its classifier by another.

- Provides means for turning marking interaction ON and OFF in the associated widget.
- Provides means to turn ON and OFF, the recognition of a particular mark in the widget.
- Displays the representative shapes of the mark classes classified by the widget.

4.1.2 Making Marks in a Widget

One makes a mark in a widget by using the mouse. A mark is made by pressing the pre-specified mouse button (+ key) combination, drawing the mark in the widget by moving the mouse, and then releasing the button (+keys) when done. By default, the middle mouse button is assigned for the purpose.

4.1.3 Controlling Marking Interaction in a Widget

To activate (deactivate) marking interaction, and to view the representative examples of marks classified in a widget, one uses different key (+button) combinations. The default settings are "<Alt>a" for activating and "<Alt>d" for deactivating marking interaction in a widget. "<Alt>h" is the default setting for viewing the marks. The marks are displayed in a transient scrolled window, each one on a toggle button. To activate (deactivate) a particular mark, one simply needs to click on its button. An activated mark is displayed in a brighter color, whereas, a deactivated mark is displayed in a darker color. The state of a mark toggles, each time its button is pressed.

Here is a summary of default key (button) settings used in a `MarkInterface` object:

- "middle mouse button" for making marks in a widget.
- "<Alt>a" for activating marking interaction in a widget
- "<Alt>d" for deactivating marking interaction in a widget.
- "<Alt>h" for viewing marks recognized in a widget.

Information regarding a classified mark is passed to the application (as a parameter of the call-back function) in a structure called **`MarkEvent`**. This data structure is similar to that of the `XEvent` structure in X. Following is the declaration of the MarkEvent structure.

```
typedef struct {
   Widget        widget;                 // widget where mark was made
   Time          start_time, end_time;   // start and end time of mark
   int           start_x, start_y;       // starting point of mark
   int           last_x, last_y;         // ending point of mark
   unsigned int  state;                  // state of other keys (buttons)
   MarkPoint     *path_points;           // points in path of mark
   unsigned int  npoints;                // number points in mark
   char          *mark_class_name;       // class name of the mark
   MarkBBox      bbox;                   // bounding box of the mark
} MarkEvent;
```

4.2 Steps in augmenting a Motif Widget with Marking Interaction

1. Include the "**MarkInterface.H**" file in the application source.
2. Create the "toplevel" shell object in your application. After this, call the function "**MarkAppInitialize()**" passing it the "XtAppContext" object in your application as the only parameter. This call initializes the Mark package and does some initial set up work essential for providing mark-based interaction.
3. Instantiate a MarkInterface object by passing to its constructor, a pointer to the widget as the first parameter, and either a pointer to a pre-created classifier, or the name of the classifier file as the second parameter. The second parameter is optional and can be left out. MarkInterface provides a method to set the classifier at a latter time. However, until a classifier is provided to a MarkInterface object, it wouldn't be able to support marking interaction.
4. Register a call-back procedure with the MarkInterface object. Although this step is optional, without a call-back, MarkInterface would have no way of contacting the application upon recognition of a mark. An optional user data can be supplied when registering the call-back. Similar to Motif, this data is passed back with the call-back function.
5. If one wants to change the default key (button) combinations for making marks in a widget, or to control marking interaction in widget, one needs to set an environment variable "**MARKUSERTRANSLATIONS**" to any integer value. It does not matter to what value the variable is set, as long as it is set. This is to indicate to a MarkInterface object to use the user defined key (+ button) combinations for the above described purposes, rather than the default combinations. The user must provide his (her) translations in an "default-resources" file located in the appropriate place. By default, a "default-resources" file is present in the "/usr/lib/X11/app-defaults/" or "/usr/openwin/lib/app-defaults/" directory. If one wants to place this file in some other place, the **XAPPLRESDIR** environment variable must be set to the absolute path of the new place. The path must end with a slash "/".

   The default translation table settings for the various actions are:

```
<Btn2Down>:     btnDownAction()      // starts a mark
<Btn2Up>:       btnUpAction()        // ends a mark
<Btn2Motion>:   btnMotionAction()    // collects points of a mark
Alt<Key>a:      activateAction()     // activates marking
Alt<Key>d:      deactivateAction()   // deactivates marking
Alt<Key>h:      helpAction()         // displays marks classified
```

   The action names are on the right side of the table. To change these default translations, one must use the same action names as given in the right column of the table above.
6. Link the application code with the mark library "**libmark.a**".

5.0 Conclusion

A system is provided to augment Motif widgets with mark-based interaction without affecting their normal functionality and without having to modify any internal Motif code. Interaction using marks can be supported in a Motif widget by inserting a layer or interface between the widget and the application. The interface object is developed as a part of this system and is termed `MarkInterface`. Mark-based interaction requires classifiers to distinguish between the possible marks a user may make. A `MarkInterface` object embodies a classifier and the code to trap and process all the marking related events in the widget. One instantiates a `MarkInterface` object by passing it the widget one wishes to augment, and a classifier. One may also register a call-back function with the interface object which is called upon recognition of a mark made in the widget. The call-back function is passed a structure of information (termed `MarkEvent`) about the classified mark which can be used by the application for any further processing. Besides recognition of marks, the other functions supported by a `MarkInterface` object are: displaying representative shapes of marks classified it, activation and deactivation of marking interaction in the widget, and activation and deactivation of particular marks in the widget. Specific key (+ button) sequences are ascribed to invoke these functions. These settings can however be changed to suit the user's preferences simply by setting an environment variable. In addition, the classifier associated with a `MarkInterface` object can also be changed at any point in the application code by invoking a method of the interface.

A stand-alone application "`MarkEditor`" is provided as the second part of this system to facilitate creation and maintenance of classifiers (termed as `MarkClassifiers`) to be used in a `MarkInterface` object. Elementary statistical pattern recognition techniques are used to produce mark classifiers that are trained by example, greatly simplifying their creation and maintenance. The main functions supported are: creation of a new classifier, addition of marks and new mark classes to a classifier, removal of classes from a classifier, setting the representative shape of a class, classifying examples made by the user (test driving a classifier), saving a classifier in a file, and opening and modification of an existing classifier. To facilitate quick creation and change of classifiers by reusing marks, the editor provides a mechanism to create and maintain inventories of marks termed as `MarkCatalogs`. To create a new classifier, or add new classes to an existing classifier, one can simply choose classes of marks from a catalog and add them to the classifier. Functions are supported to add and remove classes from a catalog, add new examples to a class, view and set representative shape of a class, view mark examples in a class, and open and save a catalog.

Striking the right balance between mark-based and direct-manipulation interaction in an application, in order to enhance the overall ease of use and appeal of an application, may require an iterative effort (which may utilize end-user feedback) on the developer's part, and can only be perfected with experience. However, it is hoped that this system, because of its simplicity and ease of use, will encourage developers to experiment with mixing the above styles of interaction techniques in a Motif application.

6.0 Bibliography

1. Duda R. and Hart P. *Pattern Classification and Scene Analysis*. Wiley Interscience, 1973.
2. Goldberg David and Richardson Cate. "Touch-Typing With a Stylus." *Proceedings of INTERCHI'93*, pp 80-87, April 1993.
3. Heller Dan. *Motif Programming Manual for OSF Motif Version 1.1*, Vol. 6, O'Reilly & Associates, Inc., 1991.
4. James Mike. *Classification Algorithms*. Wiley-Interscience. John Wiley and Sons, Inc., New York, 1985.
5. Kimura Takayuki Dan. "Hyperflow: A Uniform Visual Language for Different Levels of Programming." *Proceedings of ACM Computer Science Conference*, Indianapolis, Indiana 1993.
6. Kurtenbach and Buxton William. "Issues in Combining Marking and Direct Manipulation Techniques." *Proceedings UIST'91*: pp 137-144, November 1991.
7. Linton Mark A., Vlissides John M., and Calder Paul R. "Composing User Interfaces with InterViews." *IEEE Computer*: pp 8-22, February, 1989.
8. Lipscomb James S. "A Trainable Gesture Recognizer." *Pattern Recognition*, 24(9): 895-907, 1991.
9. Nye Adrian. *Xlib Programming Manual for Version 11*, Vol. 1, O'Reilly & Associates, Inc., 1989.
10. Nye Adrian and O'Reilly Tim. *X Toolkit Intrinsics Programming Manual for Version 11*, Vol. 4, O'Reilly & Associated, Inc., 1993.
11. Rubine Dean H. "The Automatic Recognition of Gestures." *Ph.D. Thesis, CMU-CS-91-202, Computer Science, Carnegie Mellon University*, Pittsburgh, PA, 1991.
12. Taysi Burak M. "Gesture System for a Graph Editor." *Master's Thesis. Sever Institute, Washington University*, St. Louis, MO, 1992.
13. Vlissides John M. and Linton Mark A. "Unidraw: A Framework for Building Domain-Specific Graphical Editors." *ACM Transactions on Information Systems*, 8(3): 237-266, July 1990.
14. Wolf, C.G. "A Comparative Study of Gestural, Keyboard, and Mouse Interfaces." *Behavior & Information Technology*, 11(1): 13-23, 1992.
15. Zhao Rui. "Incremental Recognition in Gesture-Based and Syntax-Directed Diagram Editors." *Proceedings INTERCHI'93*: pp 95-100, April 1993.
16. Zhao Rui. "On-line Geometry Recognition using C++, an Object-Oriented Approach." *In Proceedings of the 7th International conference & Exhibition of Technology of Object-Oriented Languages and Systems, TOOLS*, 7:371-378, April 1992.

Appendix A

Code for the Marks package:

This appendix contains the actual C++ code used in constructing the MarkInterface, the MarkEditor, MarkClassifier and MarkCatalog objects described earlier in this report. The code for the following files is provided:

1. MarkMisc.H

   Contains some miscellaneous definitions and declarations.

2. MarkExample.H, MarkExample.C

   Class that abstracts a mark example.

3. `MarkExampleSet.H, MarkExampleSet.C`

   Class that abstracts a set of mark examples.

4. `MarkCatalog.H, MarkCatalog.C`

   Class that abstracts a catalog of marks.

5. `MarkClassifier.H, MarkClassifier.C`

   Class that abstracts a mark classifier.

6. `MarkEditor.H, MarkEditor.C MarkEditorMain.C`

   Class that implements the functionality of a mark editor. The editor facilitates creation and maintenance of classifiers and catalogs.

7. `MarkDefaults.H`

   Contains default settings for the user interface of the editor.

8. `MarkInterface.H, MarkInterface.C`

   Class that acts as an interface between a Motif widget and the user application and actually provides marking capabilities to the widget.

The code used for training and creating statistical classifiers was that developed by Dean Rubine (of Carnegie Mellon University) and was used with minor modifications. This code is not listed in this appendix and can be obtained free of charge via anonymous ftp from "`emsworth.andrew.cmu.edu` (subdirectory gestures). The files used in the Marks package from Rubine's code are:

`sc.h, sc.c, fv.h, fv.c, matrix.h, matrix.c, util.h, util.c, bitvector.h,` and `bitvector.c`.

```
/*****************************************************************************/
/* MarkCatalog.H                                                             */
/*                                                                           */
/* Description of class MarkCatalog                                          */
/*                                                                           */
/* Author:  Lalit K. Agarwal, Brown University                               */
/* Date  :  May 1994                                                         */
/*****************************************************************************/

#ifndef MARKCATALOG_H
#define MARKCATALOG_H

#include "MarkExampleSet.H"

#define CATALOG_MAX_CLASSES  100

/*****************************************************************************/
/* MarkCatalog                                                               */
/*                                                                           */
/* Class to create, store, browse, and manipulate a collection of Marks      */
/* (or called MarkExamples) of different classes. This collection can then   */
/* be used to create Classifiers (MarkClassifiers). The following basic      */
/* functions are supported by this class:                                    */
/*                                                                           */
/* 1) Adding examples (MarkExamples) of existing and new classes.            */
/* 2) Setting representative examples of existing and new classes.           */
/* 3) Reading and writing the catalog from/to a file.                        */
/* 4) Retrieving examples of a given class.                                  */
/* 5) Retrieving rep. examples of all the classes in the catalog.            */
/* 6) Retrieving rep. example of a given class.                              */
/* 7) Removing classes from a catalog.                                       */
/*                                                                           */
/* NOTE: All the classes are distinguished only on the basis of their names. */
/*****************************************************************************/

class MarkCatalog
{
private:

  char             *_catalogFile;                    // catalog file
  uint             _nclasses;                        // num classes
  char             *_classNames[CATALOG_MAX_CLASSES]; // array of class-names

  MarkExampleSet   _exampleSetArray[CATALOG_MAX_CLASSES]; // actual examples
  int              _changed;                         // status


  //    HELPER METHODS

  //    if class-name is new, returns new index;
  int
  class_index (const char *className);


  //
  void
  set_class_name (const char *newName, uint index);


public:

  //    CONSTRUCTOR
  //
  MarkCatalog (const char *file = NULL);


  //    DESTRUCTOR
  //
  ~MarkCatalog();


  //   ACCESSOR functions


  //
  int
  changed() const
        { return _changed;}


  void
  set_change_status(int new_status)
        { _changed = new_status;}


  //
  const char*
  catalog_file () const
        {return _catalogFile;}


  //
  void
  set_file_name(const char *new_name);


  //    returns an array of names of the classes of marks present
  //    in the catalog. returns NULL on failure.
  //
  const char**
  class_names  () const;


  //
  uint
  nclasses () const
                {return _nclasses;}

  //    returns 1 if catalog empty, 0 if not.
  short
  empty() const
                {return !_nclasses;}


  //   BROWSING CATALOG

  //   given class-name, returns index; returns -1 if class-name doesn't exist
  //
  int
  name_to_index (const char* name) const;


  //    tells if the given class name is present in the catalog.
  //    returns 1 if class present, 0 if not.
  int
  class_member(const char *name) const
  { return (name_to_index(name) >= 0) ? 1 : 0 ; }


  //   given class index or classname, return its examples-set
  //   if invalid index, returns NULL. The returned example-set can be
  //   changed.
  //
  MarkExampleSet*
```

```
  example_set (uint index) ;

  //    same as above method; however, it also handles new class-names.
  //
  MarkExampleSet*
  example_set (const char *className);


  //   return rep-examples of all the classes in this catalog.
  //   CREATES an array of MarkExamples, and inits that with the rep-examples.
  //   It is the responsibility of the calling function to appropriately FREE
  //   the memory allocated for the returned array of MarkExamples when done.
  //
  MarkExample*
  rep_examples(int *nexamples) const;


  //   given index or classname, returns pointer to its representative example
  //
  const MarkExample*
  rep_example (uint index) const;

  const MarkExample*
  rep_example (char *className) const;



  //   EDITING CATALOG

  int
  read_catalog ();                      // re-reads current catalog

  //
  int
  read_catalog (const char *fileName);  // reads a new catalog

  void
  write_catalog()
        { write_catalog (_catalogFile);} // writes the current catalog

  void
  write_catalog (const char *fileName);  // writes catalog to given file

  //    Add a new example "newExample" of "className" to the catalog.
  //    Can add examples of both existing and new classes.
  //
  void
  add_example (const char* className, const MarkExample &newExample);

  //    Set representative example of a class. Again, does that for
  //    both new and existing classes.
  //
  void
  set_rep_example (const char* className, const MarkExample &repExample);


  //
  int
  remove_class (const char* className);

  //    deletes _catalogFile, cleans up the "_classNames" array and the
  //    exampleSets. Makes _nclasses equal 0.
  void
  cleanup();

); //end MarkCata

#endif
```

```
/*****************************************************************************/
/* MarkClassifier.H                                                          */
/*                                                                           */
/* Description of class MarkClassifier                                       */
/*                                                                           */
/* Author:  Lalit K. Agarwal, Brown University                               */
/* Date   :  May 1994                                                        */
/*****************************************************************************/

#ifndef MARCLASSIFIER_H
#define MARCLASSIFIER_H

#include <stdio.h>
#include <stdlib.h>
#include <iostream.h>

#include "MarkMisc.H"



extern "C" {
#include "../recog/bitvector.h"
#include "../recog/matrix.h"
#include "../recog/util.h"
#include "../recog/sc.h"
#include "../recog/fv.h"
}

#define CLASSIFIER_NUMFEATURES     13
#define CLASSIFIER_MAXCLASSES     100
#define MAHALANOBISDIST_THRESHOLD 144
#define AMBIGUITY_THRESHOLD       0.99

class MarkCatalog;
class MarkExample;

typedef enum {RECOG_DISABLED, RECOG_ENABLED} RecogStatus;


/*****************************************************************************/
/* Class MarkClassifier                                                      */
/*                                                                           */
/* Linear-discrimination-based statistical classifier. Classifies a given    */
/* mark into one of the constituent classes. Before classification, the      */
/* classifier must be trained with examples of each class.                   */
/* A given example is first converted into a feature vector (of 12 features) */
/* which is then classified.                                                 */
/*                                                                           */
/*****************************************************************************/

class MarkClassifier
{
private:

  sClassifier   _sc;                                    // classifier data
  char          *_classNames[CLASSIFIER_MAXCLASSES];    // class names
  MarkExample   _repExamples[CLASSIFIER_MAXCLASSES];    // rep examples
  char          *_fileName;                             // file name
  int           _changed;                               // status
  RecogStatus   _classStatus[CLASSIFIER_MAXCLASSES];

  //    HELPER METHODS
  //
  int             index (const char *className) const;

  //    example to feature vector
  //
  Vector          example_to_fv (const MarkExample &ex) const;

public:

  //    CONSTRUCTOR
  //
  MarkClassifier (const char *file = NULL);

  //    DESTRUCTOR
  //
  ~MarkClassifier();

  //   ACCESSOR FUNCTIONS

  int
  changed() const
        { return _changed;}

  void
  set_change_status(int new_status)
        { _changed = new_status;}

  int
  nclasses() const
        { return _sc ? _sc->nclasses : 0; }

  //
  const
  char* file_name() const
        { return _fileName;}

  void
  set_file_name(const char *new_name);

  //
  int
  nfeatures() const
        { return _sc ? _sc->nfeatures: 0;}

  //
  const char **
  class_names() const;

  //    returns 1 if classifier trained, 0 if untrained.
  int
  trained() const
        { return (_sc->w) ? 1 : 0; }


  //   BROWSING

  int
  read_classifier (const char* infile);

  void
  write_classifier (const char* outfile);

  void
  write_classifier()
        { write_classifier(_fileName);}


  //    returns the array of representative examples of all the
  //    mark classes recognized by this classifier. The input parameter
  //    "num" is set to the number of elements in the returned array.
  //    Returns NULL on failure.
  //
```

```
  const MarkExample*
  rep_examples (int *num);

  //    returns pointer to the representative example of the input classname
  //    return NULL on failure (nonexistant or null classname)
  //
  const MarkExample*
  rep_example (const char *className);

  //    same as the above function.
  //
  const MarkExample*
  rep_example(int class_index);


  //   EDITING

  //    add the example "newExample" of class name "className" to the
  //    classifier. Accepts examples of both existant and new class names.
  //
  void
  add_example (const char *className, const MarkExample &newExample);

  //    same as above method except that given a catalog, and a list of
  //    class names from the catalog, it adds all the examples of each class
  //    in the catalog to the classifier.
  //
  void
  add_examples (const char *classNames[], int nclasses,
                MarkCatalog *catalog);

  //    Removes a class of the given name from the classifier. Re-trains the
  //    classifier after doing so.
  int
  remove_class(const char *className);

  //    sets the representative example of class only if it is exists.
  //    doesn\'t set the rep. example of a new class.
  //
  void
  set_rep_example (const char *className, const MarkExample &example);

  //    train classifier on the added examples
  //
  void
  train();

  //    classify a feature vector. Sets the rejection parameters
  //    ap and dp. ap is the ambiguity metric. dp is the mahalanobis distance.
  //    Threshold values are defined for both the metrics. The higher the value
  //    of ap for an example, the better it is, and the reverse is true for dp.
  //
  const char *
  classify_fv(Vector fv, double *ap, double *dp) const;

  //
  const char *
  classify_fv(Vector fv) const;

  //    classifies the given example and returns it\'s class name.
  //    returns NULL if example doesn\'t belong to any class.
  //
  const char *
  classify_example(const MarkExample *ex_to_classify) const;

  //    classifies all the examples of each class whose names are given
  //    in the input parameter (array "classNames[]").
  //
  void
  classify_examples(char *classNames[], int nclasses,
                    MarkCatalog *catalog);

  //    TO DO
  void remove_mark_class (uint index) {;}
  void remove_mark_class (char* className) {;}

  //    Toggles the recognition status of a particular mark class
  //    between ENABLED and DISABLED. When the status for a class is
  //    DISABLED, the recognizer returns NULL class-name whenever an
  //    example of that class is recognized.
  //
  void toggle_recognition_status (int class_index);
  void toggle_recognition_status (const char *className);

  //    RecogStatus tells whether the recognition of a particular mark class is
  //    enabled or disabled in the classifier. If it is disabled, then if an
  //    input mark is classified as of that class, the classifier returns null.
  //
  RecogStatus  enabled(int class_index) const;
  RecogStatus  enabled(const char *class_name) const;

  //    to find out if the input class name is recognized or not by
  //    the classifier.
  //
  int   new_class(const char *class_name) const
        { return (index(class_name) < 0) ? 1 : 0; }

  //    cleans up the classifier. the names in the _classNames array,
  //    are all deleted, all the examples in the  _repExamples array are also
  //    recursively cleaned up. The classifier data "_sc" is also freed.
  //    _fileName is deleted and set to NULL.
  //
  void  cleanup();

}; //end MarkClassifier


#endif
```

```
/*****************************************************************
* FILE     :    MarkDefaults.H
*
* AUTHOR   :    LALIT AGARWAL
*
* DATE     :    May 1994
*
* This file contains fall back resources for the MarkEditor UI.
*****************************************************************/



/*****************************************************************
*                F A L L - B A C K
*                R E S O U R C E S
*
*****************************************************************/

static String fallback_res [] =
{
  "Mark.width  : 500",
  "Mark.height : 500",

  //    Fallback resources for Main Window

  "Mark.MainWindow.background  : white",
  "Mark.MainWindow.background  : black",

  //    Fallback resources for Drawing Area

  "*drawing_area.width       : 400",
  "*drawing_area.height       : 400",

  "*drawing_area.background  : grey90",
  "*drawing_area.foreground  : black",

  "*help_scrolled_win.width       : 600",
  "*help_scrolled_win.height       : 400",


  NULL

  }; //end of fallback resources
```

```
/*****************************************************************************/
/* MarkEditor.H                                                              */
/*                                                                           */
/* Description of class MarkEditor                                           */
/*                                                                           */
/* Author:  Lalit K. Agarwal, Brown University                               */
/* Date  :  May 1994                                                         */
/*****************************************************************************/

#ifndef MarkEditor_H
#define MarkEditor_H

#include "MarkCatalog.H"
#include "MarkClassifier.H"

#include <iostream.h>

#include <Xm/Xm.h>

#define UI_MAX_SELECTED_ITEMS   50
#define UI_MAX_NAMELEN         100
#define UI_MAX_CATALOGS          5
#define UI_MAX_CLASSIFIERS      25


/*****************************************************************/
/* struct MarkMenuItem                                           */
/*                                                               */
/* Used in the construction of menus for the MarkEditor          */
/*****************************************************************/

struct MarkMenuItem
{
  char          *label_str;     //widget's name
  char          mnemonic;       //button's mnemonic
  char          *accelerator;   //accelerator if any
  char          *accel_text;    //accelerator text
  WidgetClass   *class_ptr;     //pushbutton, label, separator...

  //    routine to call
  void          (*callback)(Widget, XtPointer, void*);

}; //end of struct definition

/*****************************************************************/
/* struct ListSelection                                          */
/*                                                               */
/* Used in processing the selection from various list widgets.   */
/*****************************************************************/

typedef struct
{
  char  *item;
  int    item_position;
  char  *selected_items[UI_MAX_SELECTED_ITEMS];
  int    selected_item_count;
  int   *selected_item_positions;
} ListSelection;

typedef enum (NONE, CAT_ADD_EXAMPLES, CAT_SET_REP_EXAMPLE, CAT_SHOW_EXAMPLES,
              CLASSIFIER_SET_REP_EXAMPLE, CLASSIFIER_ADD_MARKS,
              CLASSIFIER_DRAWADD_MARKS,
              CLASSIFIER_CLASSIFY_EXAMPLES) MarkEditorEditMode;



/*****************************************************************/
```

```
/* class MarkEditor                                              */
/*                                                               */
/* The class for the full functionality of an editor to:         */
/*                                                               */
/* 1) create catalogs of marks                                   */
/* 2) browse existing catalogs                                   */
/* 3) append to/modify existing catalogs                         */
/* 4) create classifiers to classify a given selection of marks  */
/* 5) browse existing classifiers                                */
/* 6) append new marks to existing classifiers                   */
/*****************************************************************/

class MarkEditor
{
  //    widgets used in the user interface
  //
  Widget toplevel_w;
  Widget main_w;
  Widget menubar_w;
  Widget form_w;
  Widget cat_label;
  Widget rec_label;
  Widget cat_list_w;
  Widget rec_list_w;
  Widget message_w;
  Widget prompt_dialog;
  Widget selection_dialog;
  Widget shell_w, file_list_w, pane_w;
  Widget da_w;

  //    Drawing area related stuff
  //
  GC     da_gc;
  Pixmap da_pixmap;
  Pixel  da_fg;
  Pixel  da_bg;
  unsigned short da_width, da_height;
  Screen  *da_screen;
  Display *da_display;
  Window   da_win;
  XFontStruct*  font_struct;            // struct to store font info
  char*         font_name;              // font used for display


  //    Catalog/Classifier related stuff
  //
  MarkCatalog _catalogs[UI_MAX_CATALOGS];
  MarkClassifier _classifiers[UI_MAX_CLASSIFIERS];
  int _ncatalogs;                       // num open catalogs
  int _nclassifiers;                    // num open classifiers

  MarkCatalog *catalog;                 // current catalog
  MarkClassifier *classifier;           // current classifier

  //    Used for a current example
  //
  char example_className[UI_MAX_NAMELEN];
  MarkExampleSet *cur_exampleSet;
  MarkExample    *cur_example;

  MarkEditorEditMode edit_mode;

  ListSelection _cat_selection;         // keeps track of current selections
  ListSelection _rec_selection;


  //    UI related HELPER METHODS
```

```
  Widget
  build_pdm (char *menu_title, char menu_mnemonic,
             MarkMenuItem* menu_items);

  void
  create_menus();

  void
  create_workArea();

  void
  create_lists();

  void
  create_drawingArea();

  void
  create_messageArea();


public:

  MarkEditor (Widget toplevel);

  //    COMMON METHODS (used both for catalogs and classifiers)
  //    involve creation of a "prompt dialog".

  void
  prompt_open_new(char doc_type, char open_or_new);

  void
  prompt_save_close(char list_type, char file_mode);

  void
  prompt_save_as(char doc_type);

  void
  prompt_quit();

  void
  prompt_get_className();

  void
  prompt_show_selection(char sel);

  void
  show_selection(char sel_type, char *selection);

  void
  close_selection(char sel_type, char *selection);

  void
  show_list(char listType);


  //    OTHER UTILITY FUNCTIONS
  //

  void
  saveAll(char doc_type);

  //    for any activity in the drawing area
  //
  void
  da_activity(XmDrawingAreaCallbackStruct *da_cbs);

  void clear_da();

  //    for drawing a given example in the drawing area
  void
  draw_example(const MarkExample *ex, const char *name);

  //    for printing a message in the message area.
  //
  void
  print_message (char *message, char new_or_cont);

  //    called when one selects an item(mark) from one of the
  //    two lists. The rep. example of the selected mark is displayed
  //    if present.
  void
  update_selection (Widget w, XmListCallbackStruct *cbs);

  //    clears the current selection (in the catalog/classifier lists)
  void
  clear_selection(char doctype);


  MarkEditorEditMode
  get_edit_mode()
        { return edit_mode;}

  void
  set_edit_mode(MarkEditorEditMode new_mode)
        {edit_mode = new_mode;}

  void
  save_quit();


  //    CATALOG RELATED
  void
  read_catalog(char *file_name, char open_or_new);

  void
  save_catalogs(int nitems, char **items);

  void
  write_catalog(char *file_name)
        { catalog->write_catalog(file_name);}

  void
  close_catalogs(int nitems, char **items);

  //    given the name of the catalog file, returns the (internal) slot
  //    number (or index)  it is in.
  int
  catalog_index(char *name);

  const char *
  curCatalogSelection() const
  { if (catalog)
      return _cat_selection.item;
    else
      return NULL;
  }

  //    to add examples to a catalog
  void
  catalog_addExamples(const char *class_name);

  void
  catalog_addNewClass();
```

```
  void
  catalog_removeClass();

  void
  show_examples (const char *class_name);


  //    CLASSIFIER RELATED

  void
  new_classifier();

  void
  read_classifier(char *file_name, char open_or_new);

  void
  save_classifiers(int nitems, char **items);

  void
  write_classifier(char *file_name)
        (classifier->write_classifier(file_name);)

  void
  close_classifiers(int nitems, char **items);

  int
  classifier_index(char *name);

  const char *
  curClassifierSelection() const
        { if (classifier)
            return _rec_selection.item;
          else
            return NULL;
        }

  void
  classifier_setRepExample(const char *class_name);

  //    initial setup before adding new marks to a classifer
  void
  rec_setup_addMarks();

  void
  rec_addMarks();

  void
  classifier_drawAddMarks(const char *class_name);

  void
  classifier_addNewClass();

  void
  classifier_removeClass();

  void
  rec_train();

  void
  classifier_classifyExamples();

  //    HELP TUTORIAL
  void help_tutorial();

}; /* class MarkEditor */

#endif
```

```
/*****************************************************************************/
/* MarkExample.H                                                             */
/*                                                                           */
/* Description of class MarkExample                                          */
/*                                                                           */
/* Author:  Lalit K. Agarwal, Brown University                               */
/* Date   :  May, 1994                                                       */
/*****************************************************************************/

#ifndef MARKEXAMPLE_H
#define MARKEXAMPLE_H

#include "MarkMisc.H"

#define EXAMPLE_DEFAULT_NUMPOINTS 100


/*****************************************************************************/
/* class MarkExample                                                         */
/*                                                                           */
/* Class to store the points, and name of a Mark(or simply, example).        */
/*****************************************************************************/

class MarkExample
{
  char          *_name;         // example name if any
  uint          _npoints;       // # actual points stored in "_pointArray"
  MarkPoint     *_pointArray;   // array of points
  uint          _size;          // size of "_pointArray"


  //    HELPER METHODS

  uint
  size() const
        { return _size;}

  //
  void
  init_name (const char *initName);

  //
  void
  init_array (uint size);


public:

  //    can create and initialize at the same time
  //    also acts as the DEFAULT CONSTRUCTOR
  //
  MarkExample(uint npoints = EXAMPLE_DEFAULT_NUMPOINTS,
              MarkPoint *initPoints = NULL,
              const char* initName = NULL);

  //    MEMBERWISE-INITIALIZATION CONSTRUCTOR
  //
  MarkExample (const MarkExample &rhs);

  //    DESTRUCTOR
  //
  ~MarkExample();


  //    assignment operator "="
  //
  MarkExample&
  operator = (const MarkExample &rhs);

  //    overwrites the example with a new one: doubtfully needed
  //
  void
  over_write (uint npoints, MarkPoint *points, char *newName = NULL);


  //    adds a point and on success, returns the total number of actual
  //    points in the example, including the new one just added.
  //    On failure, returns 0
  //
  int
  add_point(const MarkPoint& newPoint);


  //    ACCESSOR METHODS

  //
  int
  empty () const
        { return !_npoints;}

  //
  uint
  npoints() const
        { return _npoints;}

  //
  const char*
  name() const
        { return _name;}

  //
  void
  set_example_name (const char *newName);

  //    if valid index, assigns the point to inPoint
  //    On Success, returns the index of the point
  //    On failure, returns -1;
  //
  int
  get_point (MarkPoint &inPoint, uint index) const;


  //    returns pointer to point at given index, returns NULL on failure
  //
  MarkPoint*
  get_point (uint index) const;


  //    creates (using operator new) and returns a new copy of points.
  //    returns pointer to the array of MarkPoints, sets argument "npoints"
  //    to the size of the returned array.
  //    Returns NULL (with npoints set to 0) if example is empty.
  //
  MarkPoint*
  points (uint *npoints) const;

  //    sets "inbox" to the bbox values of the example.
  //    returns 1 if "inbox" set to a valid value, else returns 0.
  int
  bbox(MarkBBox &inbox) const;

  //    Deallocates space for the old array of points and initializes space
  //    for the new (future) points.
  void
```

```
  cleanup();

}; //MarkExample


#endif
```

```
/*****************************************************************************/
/* MarkExampleSet.H                                                          */
/*                                                                           */
/* Description of class MarkExampleSet.                                      */
/*                                                                           */
/* Author:  Lalit K. Agarwal, Brown University                               */
/* Date  :  May 1994                                                         */
/*****************************************************************************/

#ifndef MARKEXAMPLESET_H
#define MARKEXAMPLESET_H

#include <assert.h>

#include "MarkExample.H"

#define EXAMPLESET_DEFAULT_NUMEXAMPLES 10

/*****************************************************************************/
/* class MarkExampleSet                                                      */
/*                                                                           */
/* This class represents a collection of MarkExamples. It also has space to  */
/* store an index (user defined), a name, and a representative example for   */
/* the collection if desired. The user can think of it as an advanced array  */
/* of MarkExamples.                                                          */
/*****************************************************************************/

class MarkExampleSet
{
  int            _classIndex;      // index of the class
  char           *_className;      // name of the mark class
  uint           _nexamples;       // actual # examples present
  MarkExample    *_repExample;     // representative example
  MarkExample    *_exampleArray;   // array of examples
  uint           _size;            // size of the example-array

  //    HELPER METHODS

  uint
  size() const
      { return _size;}

  void
  init_array (uint size)
       { _size = size;
         _exampleArray = new MarkExample [_size];
         assert (_exampleArray);
       }

  void
  init_name (const char *initName);

public:

  //    CONSTRUCTOR
  //    Also acts as the default constructor
  //
  MarkExampleSet (uint nexamples = EXAMPLESET_DEFAULT_NUMEXAMPLES,
                  const MarkExample *initArray = NULL,
                  int index = -1,
                  const char *initName = NULL);


  //    MEMBERWISE-INITIALIZATION CONSTRUCTOR
  //    called during: initialization by assignment, pass-by-value,
  //                   and return-by-value
  //
  MarkExampleSet (const MarkExampleSet &rhs);


  //    DESTRUCTOR
  //
  ~MarkExampleSet();


  //    ASSIGNMENT OPERATOR "="
  //
  MarkExampleSet&
  operator = (const MarkExampleSet &rhs);


  //   ACCESSOR FUNCTIONS
  //
  int
  index() const
       { return _classIndex;}

  //
  void
  set_index(int newIndex)
       { _classIndex = newIndex;}

  //
  uint
  nexamples() const
       { return _nexamples;}

  //    returns 1 if empty, 0 otherwise.
  short
  empty() const
       { return !_nexamples;}

  //
  const char*
  class_name() const
       { return _className;}

  //
  void
  set_class_name (const char *newName);


  //    Creates an array of MarkExamples, initializes it with examples
  //    present in the set, and returns a pointer to the array. Also
  //    sets the input operand to the size of the array.
  //
  MarkExample*
  get_all_examples(uint *nexamples) const;


  //    given index, sets the operand "inExample" to the example at that index.
  //    Returns the index if its valid, otherwise, returns -1.
  //
  int
  get_example(MarkExample &inExample, uint index) const;


  //    given index, returns pointer to the example.
  //    returns NULL upon failure. The returned example can be modified.
  //
  MarkExample*
  get_example(uint index) const;
```

```
  //    Appends the new example to the set.
  //    Returns the total # of examples in the set including the new one.
  //
  uint
  add_example(const MarkExample& newExample);


  //
  const MarkExample*
  rep_example() const
        { return _repExample;}

  //
  void
  set_rep_example(const MarkExample &newExample);

  //    deallocates space for the old examples, rep-example and class-name.
  //    also initializes the space for storing new examples.
  void
  cleanup();


}; //class MarkExampleSet


#endif
```

```
/*****************************************************************************/
/* MarkInterface.H                                                           */
/*                                                                           */
/* Description of class MarkInterface                                        */
/*                                                                           */
/* Author:  Lalit K. Agarwal, Brown University                               */
/* Date  :  May 1994                                                         */
/*****************************************************************************/


#ifndef MARK_INTERFACE_H
#define MARK_INTERFACE_H

#include <Xm/Xm.h>

class MarkInterface;
class MarkClassifier;
class MarkExample;

#include "MarkMisc.H"


/*****************************************************************************/
/*              struct MarkEvent                                             */
/*                                                                           */
/* This structure is passed to the application callback routine when a mark  */
/* -- made by the user in the widget -- is recognized.                       */
/*****************************************************************************/

typedef struct
{
  Widget        widget;                 // widget in which mark was made
  Time          start_time, end_time;   // start and end time of mark
  int           start_x, start_y;       // starting point of mark
  int           last_x, last_y;         // ending point of mark
  unsigned int  state;                  // state of other keys/buttons
  MarkPoint     *path_points;           // points in the path of the mark
                                          // user should "delete" this array after
                                          // being done with it.

  unsigned int  npoints;                // number points in the mark
  char          *mark_class_name;       // unique class-name of the mark
  MarkBBox      bbox;                   // bounding box of the mark

} MarkEvent;


/*****************************************************************************/
/*              Forward  Function Declarations                               */
/*****************************************************************************/

//      Any application intending to use the MarkInterface, should call this
//      function before instantiating any objects of type MarkInterface.
//      This call can be made after user creates the Xt application context in
//      his application.

extern void    MarkAppInitialize(XtAppContext app_context);

//      Function to initialize a MarkEvent object
extern void    initMark(MarkEvent &mark_event);

/*****************************************************************************/
/*              Typedef for Callback Proc                                    */
/*****************************************************************************/

//      Prototype for call back procedure which the user expects to be called
//      by a MarkInterface object upon recognition of a mark.

typedef void (*MarkCallbackProc) (MarkInterface *mi,
                                  void *user_data,
                                  MarkEvent *rec_mark);


/*****************************************************************************/
/*              class MarkInterface                                          */
/*                                                                           */
/* This class is used to enhance a Motif widget with Marking-interaction     */
/* capability. To enable a widget to support marking (or gesture) interaction*/
/* one instanciates an object of this class and links it up with the widget. */
/* When instanciating this class, one needs to specify the widget it is to   */
/* be linked with, and optionally, a classifier or the file in which it is   */
/* stored. The classifier forms the core of the recognition task. A classifer*/
/* classifies a given mark into one of the classes it is trained to classify.*/
/* Classifiers can be created and stored in files using an accompanying      */
/* editor "MarkEditor". Details about this can be found in file "MarkEditor.C*/
/* A MarkInterface object without a classifer would not be able to support   */
/* marking interaction. In addition to the widget and the classifier, one    */
/* also needs to specify a callback procedure -- just like callback routines */
/* in Motif widgets -- which is called upon recognition of a mark.           */
/* Anyway, here are some of the important functionality supported by this    */
/* class:                                                                    */
/*                                                                           */
/* 1) Support marking-based interaction with Motif widgets.                  */
/*    i) Traps all the marking related mouse+key events from the widget      */
/*       from the start of a mark until its end. The mark is then classified */
/*       using the classifier within.                                        */
/* 2) Upon successful recognition of a mark, call the user specified callback*/
/*    routine. The callback routine is passed back i) a pointer to the       */
/*    MarkInterface object, the user data -- the user can optionally specify */
/*    a data when registering the callback routine (just like in Motif) --   */
/*    and a pointer to a structure of type MarkEvent. This structure contains*/
/*    all the necessary information about the recognized mark.               */
/* 3) Displays the current set of marks recognized by the classifier. This   */
/*    help can be obtained simply by pressing the prescribed sequence of keys*/
/*    in the widget.                                                         */
/* 4) Activate and deactivate marking interaction in the widget by pressing  */
/*    a key/button sequence.                                                 */
/* 5) Activate/deactivate particular marks in a widget with a key/btn seq.   */
/* 6) Change the classifier at any time, thus providing the flexibility of   */
/*    attaching a completely different set of marks to a widget during run   */
/*    time.                                                                  */
/*                                                                           */
/* One makes a mark in a widget by using the mouse. A mark is made by        */
/* pressing the specified mouse button (along with any specified keys),      */
/* draws the shape of the mark in the widget and releases the button.        */
/* By default, the middle mouse button is assigned for the purpose.          */
/* However, the user can change that by specify whatever key/button          */
/* sequence he likes in an "app-defaults" file. To do that, one uses the     */
/* same technique as of specifying a new translation-table for a widget      */
/* using the "app-defaults" file. In order to let a MarkInterface object use */
/* the user-specified translations, one must define and set an environment   */
/* variable "MARKUSERTRANSLATIONS". It can be set to any integer value. When */
/* this environment variable is defined, a MarkInterface object will expect  */
/* that the required translations are correctly defined in an "app-defaults" */
/* file which is present in the right place. By default, an app-defaults file*/
/* is present in /usr/lib/X11/app-defaults or /usr/openwin/lib/app-defaults. */
/* If you want to use a different directory, you must set the XAPPLRESDIR    */
/* environment variable to the absolute path of that directory. The path     */
/* must end with a slash "/".                                                */
/* In addition, there are key/button sequences for activating/deactivating   */
/* marking interaction in a widget, and also for displaying the mark shapes  */
/* recognized(supported) in a widget.                                        */
/*                                                                           */
/*  The default translation table for these actions is:                      */
```

```
/*                                                                          */
/*   <Btn2Down>:          btnDownAction()     \n\ // starts a mark        */
/*   <Btn2Up>:            btnUpAction()       \n\ // end of a mark         */
/*   <Btn2Motion>:        btnMotionAction()   \n\ // collects pts of mark  */
/*   Alt<Key>a:           activateAction()    \n\ // activate marking      */
/*   Alt<Key>d:           deactivateAction()  \n\ // deactivate marking    */
/*   Alt<Key>h:           helpAction()            // display marks          */
/*                                                                          */
/* To change these default translations, one must use the same action names. */
/*                                                                          */
/****************************************************************************/

class MarkInterface
{
 private:
  Widget                _widget;              // Motif widget

  MarkCallbackProc      _appCallbackProc;     // User callback routine
                                              // called upon recognition of
                                              // a mark.

  void                  *_userData;           // Optional user data passed
                                              // as a parameter to the
                                              // above callback routine.

  MarkClassifier        *_classifier;         // Classifier to perform
                                              // recognition of user made
                                              // marks in the widget.

  MarkExample           *_curUserExample;     // Mark example to be
                                              // classified.

  //    to keep track of whether marking is on or off in the widget.
  //
  enum {MARKING_DEACTIVATED, MARKING_ACTIVATED} _interactionStatus;

 public:

  //    CONSTRUCTORS

  MarkInterface(Widget w, char *classifier_file = NULL);
  MarkInterface(Widget w, MarkClassifier *classifier = NULL);

  //
  Widget get_widget() const;

  //
  const MarkClassifier*
  get_classifier() const;

  //    Allows the developer to set the classifer after instantiating
  //    a MarkInterface object.
  //
  void
  set_classifier(const MarkClassifier *new_classifier) {;}

  void
  set_classifier(const char* new_classifier_file) {;}

  //
  MarkCallbackProc
  get_callbackProc() const;

  //    The developer should set the callback proc after instantiating a
  //    MarkInterface object, if he wants MarkInterface to contact the
  //    application when a mark is recognized.
  //
  void
  set_callbackProc(MarkCallbackProc new_callback, void *new_data = NULL);

  //
  void *
  get_userData() const;

  void
  set_userData(void *new_data);

  //    Internally used. Called by an action routine at the start of
  //    a mark, during when the mark is made, and at the end of the mark.
  //
  void eventHandler(Widget w, XEvent *event);

  //    Method to display the marks recognized by the interface.
  //
  void displayMarksRecognized();

  //    Method to draw a mark. Used by the method above.
  //
  void drawExample(Widget db_w, Drawable db_win, GC db_gc, int example_index);

  //    Activated marking interaction in the widget.
  //
  void   activateMarkingInteraction();

  //    Deactivates marking interaction in the widget.
  //
  void  deactivateMarkingInteraction();

  //    To turn on and off a particular mark in the widget.
  //
  void  toggleRecognitionStatus(const char *mark_name) const;

  void  toggleRecognitionStatus(int mark_index) const;

  //    To find out if a particular mark is on or off.
  //
  int enabled(const char *mark_name) const;

  //
  int enabled(int mark_index) const;

}; /* end class MarkInterface */


#endif
```

```
/*****************************************************************************/
/* MarkMisc.H                                                                */
/*                                                                           */
/* Contains some miscellaneous definitions used throughout the package.      */
/*                                                                           */
/* Author:  Lalit K. Agarwal, Brown University                               */
/* Date  :  May 1994                                                         */
/*****************************************************************************/

#ifndef MISC_H
#define MISC_H

typedef unsigned int uint;

#define NULL 0

/*****************************************************************************/
/* struct MarkPoint                                                          */
/*****************************************************************************/

typedef struct _MarkPoint
{
  int  x, y;
  long  time;

} MarkPoint;

/*****************************************************************************/
/* struct MarkBBox                                                           */
/*****************************************************************************/

typedef struct _MarkBBox
{
  int min_x, min_y;
  int max_x, max_y;

} MarkBBox;

#endif
```

```
/*****************************************************************************/
/* MarkCatalog.C                                                             */
/*                                                                           */
/* Implementation of class MarkCatalog                                       */
/*                                                                           */
/* Author:  Lalit K. Agarwal, Brown University                               */
/* Date   :  May 1994                                                        */
/*****************************************************************************/

#include <assert.h>
#include <string.h>
#include <iostream.h>
#include <stdio.h>
#include <stdlib.h>

#include "MarkCatalog.H"

#define MAX_NAME_LENGTH 100
#define MAX_LINE_LENGTH 100
#define EXAMPLE_MAX_POINTS 1000

/*****************************************************************************/
/* Constructor: MarkCatalog(file)                                            */
/*****************************************************************************/

MarkCatalog::MarkCatalog (const char* file)
{
  _nclasses = 0;   // start fresh
  _catalogFile = NULL;
  _changed = 0;   // 0 means false

  if (file)
    read_catalog(file);

}//end MarkCatalog(file)


/*****************************************************************************/
/* Destructor                                                                */
/*****************************************************************************/

MarkCatalog::~MarkCatalog()
{
  cleanup();

} //destructor



/*****************************************************************************/
/* name_to_index(name)                                                       */
/*****************************************************************************/

int
MarkCatalog::name_to_index(const char *inName) const
{
  if (!inName) {
    // cerr << "Null string to MarkCatalog::name_to_index" << endl;
    return -1;
  }

  register short i;
  for (i = 0; i < _nclasses; i++)
    if (strcmp(inName, _classNames[i]) == NULL) return i;

  //   className not found
  return -1;

} //name_to_index()



/*****************************************************************************/
/* example_set(index)                                                        */
/*****************************************************************************/

MarkExampleSet*
MarkCatalog::example_set(uint index)
{
  if (index >= _nclasses) return NULL;

  return &_exampleSetArray[index];

} //example_set(index)



/*****************************************************************************/
/* example_set(MarkExampleSet&,index)                                        */
/*****************************************************************************/
/*
int
MarkCatalog::example_set(MarkExampleSet& inSet, uint index)
{
  if (index >= _nclasses) {
    inSet->set_index(-1);
    inSet->set_nexamples(0);
    return 0;                  // num of examples in the set
  }

  inSet = _exampleSetArray[index];  // member-wise copy
  return (int) inSet->nexamples();

} //example_set(set, index)
*/

/*****************************************************************************/
/* example_set(name)                                                         */
/*****************************************************************************/

MarkExampleSet*
MarkCatalog::example_set(const char *className)
{
  int index = class_index(className);

  MarkExampleSet* s = &_exampleSetArray[index];
  return s;

} //example_set(name)


/*****************************************************************************/
/* rep_examples(num)                                                         */
/*****************************************************************************/

MarkExample*
MarkCatalog::rep_examples(int *nexamples) const
{
  if (!_nclasses) {
    *nexamples = _nclasses;
    return NULL;
  }

  //  array of pointers
```

```
  MarkExample *reply = new MarkExample [_nclasses];
  MarkExample *ex;

  for (register short i = 0; i < _nclasses; i++) {
    ex =  (MarkExample *) rep_example(i);
    if (ex)
      reply[i] = *ex;   // copy example, use overloaded assignment operator
  }

  *nexamples = _nclasses;
  return reply;

} //rep_examples()


/*****************************************************************************/
/* rep_example(index)                                                        */
/*****************************************************************************/

const MarkExample*
MarkCatalog::rep_example(uint index) const
{
  if(index >= _nclasses) return NULL;

  const MarkExample *ex =  _exampleSetArray[index].rep_example();

  return ex;

} //rep_example(index)


/*****************************************************************************/
/* rep_example(name)                                                         */
/*****************************************************************************/

const MarkExample*
MarkCatalog::rep_example(char *name) const
{
  int index = name_to_index(name);
  if(index < 0 || index >= _nclasses) return NULL;

  const MarkExample *ex =  _exampleSetArray[index].rep_example();

  return ex;

} //rep_example(index)



/*****************************************************************************/
/* read_catalog(file)                                                        */
/*****************************************************************************/

int
MarkCatalog::read_catalog(const char* filePath)
{
  FILE  *fp;
  char line[MAX_LINE_LENGTH];
  char exampleName[MAX_NAME_LENGTH];
  char className[MAX_NAME_LENGTH];
  register int index;
  register uint npoints = 0;
  register short i;
  register char lastExampleType = '\0';
  register int x, y, t, dummy;

  MarkPoint p[EXAMPLE_MAX_POINTS];     // to temporarily store example points
  register MarkExample *ex = NULL;     // points to current example
  register MarkExampleSet *exSet = NULL; // points to current exampleSet

  if (!(fp = fopen(filePath, "r")) ) {
    cerr << "NULL file name or " << filePath << "doesn't exist." << endl;
    return 0;
  }

  // cout << "started reading " << filePath << endl;

  //    free all the existing stuff in the catalog
  cleanup();

  _catalogFile = new char [strlen(filePath) + 1];
  strcpy(_catalogFile, filePath);

  //   read the file, a line at a time
  while(1)
  {
    line[0] = '\0';
    fgets (line, MAX_LINE_LENGTH, fp);     // read one line from file

    switch (line[0])  // check line contents
    {

    case '\0':   // when reach EOF
    case 'x':    // when encounter a new example
    case 'r':    // representative example

      if(lastExampleType) //  if it's the 2nd or a later example, save it
      {
        MarkExample lastExample(npoints, p, exampleName);

        if(lastExampleType == 'r')
          exSet->set_rep_example(lastExample);
        else if(lastExampleType == 'x')
          exSet->add_example(lastExample);
        else
          cerr << "bad last example\n";
      }

      //   EOF
      if (line[0] == '\0') {
        fclose (fp);
        cout << "Ended reading Catalog: " << filePath << endl;
        _changed = 0;
        return 1;
      }

      //   START OF A NEW EXAMPLE, also marks the end of the last example
      sscanf (line, "%*s %s %s", className, exampleName);
      index = class_index (className);  // assigns a new index if needed

      //   CURRENT EXAMPLE SET (the set to which the example belongs)
      exSet = &_exampleSetArray[index];

      lastExampleType = line[0];
      npoints = 0;           // prepare for the upcoming points
      break;

    default:     //  READ A POINT

      if (sscanf(line, "%d %d %d %d", &dummy, &x, &y, &t) != 4) {
        // cerr << "bad point \n" << line << endl;
        break;
      }
      if (!lastExampleType) {
```

```
        cerr << "point without mark-name \n" << line << endl;
        fclose(fp);
        _changed = 1;
        return 0;
        // exit (1);
      }
      //   read the point
      p[npoints].x = x;
      p[npoints].y = y;
      p[npoints++].time = t;

      break;

    }// switch

  }// while


}//end read_file()



/*****************************************************************************/
/* read_catalog()                                                            */
/*****************************************************************************/

int
MarkCatalog::read_catalog()
{
  return read_catalog(_catalogFile);
}


/*****************************************************************************/
/* write_catalog(file)                                                       */
/*****************************************************************************/

void
MarkCatalog::write_catalog (const char *outFilePath)
{
  FILE *fp;

  if (!(fp = fopen(outFilePath, "w")) ) {
    cerr << "MarkCatalog::write_catalog: ";
    cerr << "null file path or can't open " << outFilePath << endl;
    return;
    // exit(1);
  }

  // cout << "started writing catalog " << outFilePath << endl;

  register MarkPoint *p;
  register MarkExample *ex;
  register const MarkExampleSet *exSet;
  register uint index, nexample, npoint;

  for(index = 0; index < _nclasses; index ++)          // for each exampleSet
  {
    exSet = &_exampleSetArray[index];

    //   write the representative example first
    if (ex = (MarkExample *)exSet->rep_example()) {
      fprintf(fp, "r %s %s\n", exSet->class_name(),
              ex->name() ? ex->name() : exSet->class_name() );
      for (npoint = 0; npoint < ex->npoints(); npoint++)
      {                                                 // for each point
        p = ex->get_point(npoint);
        fprintf(fp, "%d %3d %3d %4d\n", 0, p->x, p->y, p->time);
      }
    }
    //   write the rest of the examples in the set
    for (nexample = 0; nexample < exSet->nexamples(); nexample++)
    {
      ex = exSet->get_example(nexample);
      if(!ex) break;
      fprintf(fp, "x %s %s\n",
              exSet->class_name(),
              (ex->name() ? ex->name() : exSet->class_name()) );
      for (npoint = 0; npoint < ex->npoints(); npoint++)
      {                                                 // for each point
        p = ex->get_point(npoint);
        fprintf(fp, "%d %3d %3d %4d\n", 0, p->x, p->y, p->time);
      }
    }
  } //for(index)

  fclose(fp);
  cout << "Finished writing catalog: " << outFilePath << endl;

  //    leave status unchanged if catalog written to another file
  if (!strcmp(outFilePath, _catalogFile))
    _changed = 0;

} //write_catalog()



/*****************************************************************************/
/* add_example(name, example)                                                */
/*****************************************************************************/

void
MarkCatalog::add_example(const char *className, const MarkExample& newExample)
{
  if (!className) {
    // cerr << "catalog::add_example(): null class-name \n" ;
    return;
  }

  int index = class_index (className); // assigns a new index if needed

  _exampleSetArray[index].add_example(newExample);

  _changed = 1;

} //add_example()


/*****************************************************************************/
/* set_rep_example(name, example)                                            */
/*****************************************************************************/

void
MarkCatalog::set_rep_example(const char *className,
                             const MarkExample &repExample)
{
  if (!className) {
    // cerr << "catalog::add_example(): null className\n" ;
    return;
  }

  int index = class_index (className); // assigns a new index if needed

  _exampleSetArray[index].set_rep_example(repExample);
```

```
  _changed = 1;

} //set_rep_example()



/*****************************************************************************/
/* class_index(name)                                                         */
/*****************************************************************************/

int MarkCatalog::class_index (const char *newName)
{
  if (!newName) {
    // cerr << "Null string to MarkCatalog::class_index" << endl;
    return -1;
  }

  for (register short i = 0; i < _nclasses; i++)
    if (strcmp(newName, _classNames[i]) == NULL) return i;

  //   if "newName" is indeed new, insert it
  set_class_name(newName, _nclasses);

  //   give the set a name and index
  _exampleSetArray[_nclasses].set_class_name(newName);

  _exampleSetArray[_nclasses].set_index(_nclasses);

  _changed = 1;

  return _nclasses++;

} //class_index()



/*****************************************************************************/
/* set_class_name                                                            */
/*****************************************************************************/

void
MarkCatalog::set_class_name (const char *newName, uint index)
{
  delete _classNames[index];
  _classNames[index] = NULL;
  if (newName) {
    _classNames[index] = new char [strlen(newName) + 1];
    strcpy (_classNames[index], newName);
  }
  _changed = 1;

} //set_class_name()


/*****************************************************************************/
/* cleanup                                                                   */
/*****************************************************************************/

void
MarkCatalog::cleanup()
{
  delete _catalogFile; _catalogFile = NULL;

  for (register int i = 0; i < _nclasses; ++i) {
    _exampleSetArray[i].cleanup();
    delete _classNames[i]; _classNames[i] = NULL;
  }

  _nclasses = 0;
  _changed = 0;

} // cleanup()



/*****************************************************************************/
/* class_names()                                                             */
/*****************************************************************************/

const char**
MarkCatalog::class_names() const
{
  return _classNames;
}


/*****************************************************************************/
/* set_file_name()                                                           */
/*****************************************************************************/

void
MarkCatalog:: set_file_name(const char *new_name)
{
  if(!new_name) return;
  delete _catalogFile;
  _catalogFile = new char [strlen(new_name) + 1];
  strcpy(_catalogFile, new_name);

}


/*****************************************************************************/
/* remove_class()                                                            */
/*****************************************************************************/

int
MarkCatalog::remove_class(const char *className)
{
  //    make sure className is valid
  if (!className) return 0;
  int index = name_to_index(className);
  if (index < 0) return 0;

  //    cleanup the MarkExampleSet for the class to be removed
  //    copy the last Set into this slot if this is not the last slot

  _exampleSetArray[index].cleanup();
  delete _classNames[index]; _classNames[index] = NULL;

  int lastSlot = _nclasses - 1;

  //    copy set in the last slot into this one
  if (index < lastSlot) {
    _exampleSetArray[index] = _exampleSetArray[lastSlot];
    _classNames[index] = new char [strlen(_classNames[lastSlot]) + 1];
    strcpy(_classNames[index], _classNames[lastSlot]);

    delete _classNames[lastSlot]; _classNames[lastSlot] = NULL;
    _exampleSetArray[lastSlot].cleanup();
  }

  _nclasses--;
  _changed = 1;
```

```
  return 1;

} // remove_class
```

```
/*****************************************************************************/
/* MarkClassifier.C                                                          */
/*                                                                           */
/* Methods for  class MarkClassifier                                         */
/*                                                                           */
/* Author:  Lalit K. Agarwal, Brown University                               */
/* Date  :  May 1994                                                         */
/*****************************************************************************/

#include <string.h>
#include <math.h>

#include "MarkExampleSet.H"
#include "MarkClassifier.H"
#include "MarkCatalog.H"

#define    EPS     (1.0e-4)

#define MAX_NAME_LENGTH 100
#define MAX_LINE_LENGTH 100
#define EXAMPLE_MAX_POINTS 1000


/*****************************************************************************/
/* MarkClassifier:constructor                                                */
/*****************************************************************************/

MarkClassifier::MarkClassifier(const char *file)
{
  _fileName = NULL;
  _sc = sNewClassifier();
  _sc->cnst = NULL;
  _sc->invavgcov = NULL;

  _changed = 0;

  for (register short i = 0; i < CLASSIFIER_MAXCLASSES; ++i) {
    _classNames[i] = NULL;
    _classStatus[i] = RECOG_DISABLED;
  }
  if (file)
    read_classifier(file);

} //constructor


/*****************************************************************************/
/* destructor                                                                */
/*****************************************************************************/

MarkClassifier::~MarkClassifier()
{
  cleanup();

} //destructor()




/*****************************************************************************/
/* read_classifier()                                                         */
/*****************************************************************************/

int
MarkClassifier::read_classifier(const char *infile)
{
  if(!infile) return 0;

  FILE *fp;

  if (!(fp = fopen(infile, "r"))) {
    cerr << "read_classifier: couldn't open file: " << infile << endl;
    return 0;
  }

  //    free all the existing stuff in the classifier
  cleanup();

  _fileName = new char [strlen(infile) + 1];
  strcpy(_fileName, infile);

  _sc = sRead(fp);
  int nrows = NROWS(_sc->invavgcov);
  int ncols = NCOLS(_sc->invavgcov);
  _sc->nfeatures = nrows;

  for (int i = 0; i < _sc->nclasses; ++i) {
    _classNames[i] = new char [strlen(_sc->classdope[i]->name) + 1];
    strcpy(_classNames[i], _sc->classdope[i]->name);
    _classStatus[i] = RECOG_ENABLED;
  }

#define NEW_CLASSIFIER_READ_FORMAT
#ifdef NEW_CLASSIFIER_READ_FORMAT

  //    READ THE nexamples USED PER CLASS TO TRAIN THE CLASSIFIER
  Vector v = InputVector(fp);
  register int c;
  for (c = 0; c < _sc->nclasses; ++c)
    _sc->classdope[c]->nexamples = v[c];

  //    READ IN sumcov matrix FOR EACH CLASS
  for (c = 0; c < _sc->nclasses; ++c)
    _sc->classdope[c]->sumcov = InputMatrix(fp);

#endif

  //    READ THE REPRESENTATIVE EXAMPLES, IF ANY

  char line[MAX_LINE_LENGTH];
  char className[MAX_NAME_LENGTH];
  register int  x, y;
  register int npoints = 0;
  int ix;

  MarkPoint p[EXAMPLE_MAX_POINTS];
  register MarkExample *rep = NULL;

  while(1)
  {
    line[0] = '\0';
    fgets (line, MAX_LINE_LENGTH, fp);     // read one line from file

    switch (line[0])  // check line contents
    {

    case '\0':    // when reach EOF
    case 'r':     // representative example

      if (rep) {
        MarkExample lastExample(npoints, p, className);
        *rep = lastExample;
      }
```

```
      // EOF
      if (line[0] == '\0') {
        fclose (fp);
        printf ("\n Finished reading classifier (%s)\n", infile);
        _changed = 0;
        return 1;
      }

      //   START OF A NEW EXAMPLE, also marks the end of the last example
      sscanf (line, "%*s %s", className);
      ix = index (className);

      if (ix < 0) {
        printf("read_classifier(): unknown rep-example?\n");
        fclose(fp);
        _changed = 1;
        return 0;
      }
      rep = &_repExamples[ix];
      npoints = 0;
      break;

    default:    // READ A POINT

      if (sscanf(line, "%d %d", &x, &y) != 2) {
        // cerr << "bad point \n" << line << endl;
        break;
      }
      if (!rep) {
        cerr << "read_classifier(): point without a rep-example \n" << line << endl;
        fclose(fp);
        _changed = 1;
        return 0;
      }
      //   read the point
      p[npoints].x = x;
      p[npoints].y = y;
      p[npoints++].time = 0;

      break;

    }// switch

  }// while

} //read_classifier()



/*****************************************************************************/
/* write_classifier()                                                        */
/*****************************************************************************/

void
MarkClassifier::write_classifier(const char *outfile)
{
  if(!outfile || !_sc) return;
  if (!_sc->nclasses) {
    cerr << "Empty Classifier! Nothing written." << endl;
    return;
  }

  FILE *fp;

  if (!(fp = fopen(outfile, "w"))) {
    cerr << "Couldn't open file:" << outfile << endl;
    return;
  }

  //    CHECK FOR ANY null vectors/matrices BEFORE WRITING
  //    This situation occurs when one tries to save a new classifier
  //    after adding examples to it and without training it.

  int save_before_train = 0;

  //    The following code just creates dummy zero vector/matrices
  //    (_sc->cnst, _sc->w, _sc->invavgcov)
  //    (if the classifier being saved is untrained) for the "sWrite()" call.

  if (!_sc->cnst) {
    save_before_train = 1;
    _sc->cnst = NewVector(_sc->nclasses);
    ZeroVector(_sc->cnst);
  }
  if (!_sc->w) {
    _sc->w = (Vector *) malloc(_sc->nclasses * sizeof(Vector));
    for (int c = 0; c < _sc->nclasses; ++c) {
      _sc->w[c] = NewVector(_sc->nfeatures);
      ZeroVector(_sc->w[c]);
    }
  }
  if (!_sc->invavgcov) {
    _sc->invavgcov = NewMatrix(_sc->nfeatures, _sc->nfeatures);
    ZeroMatrix(_sc->invavgcov);
  }

  sWrite(fp, _sc);

  if (save_before_train) {
    FreeVector(_sc->cnst);
    FreeMatrix(_sc->invavgcov);
    for (int c = 0; c < _sc->nclasses; ++c)
      if(_sc->w && _sc->w[c]) FreeVector(_sc->w[c]);
    if (_sc->w) free(_sc->w);
    cerr << "Classifier: " << outfile << " saved without training!" << endl;
  }

#define NEW_CLASSIFIER_WRITE_FORMAT
#ifdef NEW_CLASSIFIER_WRITE_FORMAT

  //    WRITE THE num-examples FOR EACH CLASS

  register int c;
  register sClassDope scd;

  Vector v = NewVector(_sc->nclasses);
  ZeroVector(v);
  for (c = 0; c < _sc->nclasses; ++c) {
    scd = _sc->classdope[c];
    v[c] = scd->nexamples;
  }
  OutputVector(fp, v);
  // PrintVector(v, "nexamples = ");
  FreeVector(v);

  //    WRITE THE sumcov matrix FOR EACH CLASS

  for (c = 0; c < _sc->nclasses; ++c) {
    scd = _sc->classdope[c];
    OutputMatrix(fp, scd->sumcov);
  }

#endif
```

```
  //    WRITE THE REP-EXAMPLES NOW
  MarkExample *rep;
  MarkPoint   *p;
  int nc, np;

  for (nc = 0; nc < _sc->nclasses; ++nc)
  {
    //  write example only if non-empty
    rep = &_repExamples[nc];
    if(!rep->empty())
    {
      fprintf(fp, "r %s\n", _sc->classdope[nc]->name);
      for (np = 0; np < rep->npoints(); ++np) {
        p = rep->get_point(np);
        fprintf(fp, "%3d %3d\n", p->x, p->y);
      }
    }
  }

  fclose (fp);

  printf ("finished writing classifier (%s)\n", outfile);

  //    leave status unchanged if catalog written to another file
  if (!strcmp(outfile, _fileName))
    _changed = 0;

} //write_classifier()



/*****************************************************************************/
/* add_example()                                                             */
/*****************************************************************************/

void
MarkClassifier::add_example(const char *className,
                            const MarkExample &newExample)
{
  Vector y;

  y = example_to_fv(newExample);

  int ix = index(className);
  sAddExample(_sc, (char *)className, y);

  //    add class-name to the list if it is a new one
  if (ix < 0) {
    int i = _sc->nclasses - 1;

    //  a new class is always added at the end
    sClassDope scd = _sc->classdope[i];
    _classNames[i] = new char [strlen(scd->name) + 1];
    strcpy(_classNames[i], scd->name);
    _classStatus[i] = RECOG_ENABLED;
  }

  _changed = 1;

} //add_example()


/*****************************************************************************/
/* add_examples()                                                            */
/*****************************************************************************/

void
MarkClassifier::add_examples(const char *classNames[], int nclasses,
                             MarkCatalog *catalog)
{
  if (!catalog || !classNames) return;

  register int i, j;
  register MarkExampleSet *exs;
  MarkExample    ex;
  unsigned int nexamples;

  for (i = 0; i < nclasses; ++i)
  {
    exs = catalog->example_set(classNames[i]);
    if (!exs) continue;

    //  add all the examples in the set to the classifier
    nexamples = exs->nexamples();

    for (j = 0; j < nexamples; ++j)
      if ( (exs->get_example(ex, j)) == j)
        add_example(classNames[i], ex);

  }

  train();

} // add_examples()


/*****************************************************************************/
/* remove_class()                                                            */
/*****************************************************************************/

int
MarkClassifier::remove_class(const char *className)
{
  if (!className || !_sc->nclasses) return 0;
  int ix = index(className);
  if (ix < 0)  return 0;

  int lastSlot = _sc->nclasses - 1;

  //    CLEAN UP THIS SLOT
  delete _classNames[ix]; _classNames[ix] = NULL;
  _repExamples[ix].cleanup();
  _classStatus[ix] = RECOG_DISABLED;

  sClassDope scd = _sc->classdope[ix];
  if (scd){
    if (scd->name) free(scd->name);
    if (scd->sumcov) FreeMatrix(scd->sumcov);
    if (scd->average) FreeVector(scd->average);
    free(scd);
    _sc->classdope[ix] = NULL;
  }

  //    copy stuff from last slot if they\'re not same
  if (ix < lastSlot)
  {
    _sc->classdope[ix] = _sc->classdope[lastSlot];
    _sc->classdope[lastSlot] = NULL;
    _sc->classdope[ix]->number = ix;

    _classNames[ix] = _classNames[lastSlot];
    _classNames[lastSlot] = NULL;
```

```
    _repExamples[ix] = _repExamples[lastSlot];
    _repExamples[lastSlot].cleanup();

    _classStatus[ix] = _classStatus[lastSlot];
    _classStatus[lastSlot] = RECOG_DISABLED;
  }

  _sc->nclasses--;
  train();
  _changed = 1;

  return 1;

} // remove_class()



/*****************************************************************************/
/* set_rep_example()                                                         */
/*****************************************************************************/

void
MarkClassifier::set_rep_example (const char *className,
                                 const MarkExample &example)
{
  int ix = index(className);

  //    add rep_example only if class already exists
  if (ix >= 0) {
    _repExamples[ix] = example;
    _repExamples[ix].set_example_name(className);
    _changed = 1;
  }

} //set_rep_example()


/*****************************************************************************/
/* rep_examples()                                                            */
/*****************************************************************************/

const MarkExample*
MarkClassifier::rep_examples (int *num)
{

  if (!_sc->nclasses) {
    *num = 0;
    return NULL;
  }

/*  MarkExample *rep_array = new MarkExample[_nclasses];

  register short i;

  for (i = 0; i < _nclasses; i++)
    rep_array[i] = _repExamples[i]

*/

  *num = _sc->nclasses;
  return _repExamples;

} //rep_examples()


/*****************************************************************************/
/* rep_example()                                                             */
```

```
/*****************************************************************************/

const MarkExample*
MarkClassifier::rep_example (const char *className)
{
  if (!className) return NULL;

  int ix = index(className);

  if (ix < 0) return NULL;

  MarkExample *ex = &_repExamples[ix];

  return ex;

} //rep_example()


/*****************************************************************************/
/* rep_example()                                                             */
/*****************************************************************************/

const MarkExample*
MarkClassifier::rep_example (int class_index)
{
  if(!_sc->nclasses || (class_index >= _sc->nclasses)) return NULL;
  MarkExample *ex = &_repExamples[class_index];

  return ex;

} // rep_example()

/*****************************************************************************/
/* classify_examples()                                                       */
/*****************************************************************************/

void
MarkClassifier::classify_examples(char *classNames[], int nclasses,
                                  MarkCatalog *catalog)
{
  if (!catalog || !classNames) return;
  if (!_sc || !_sc->nclasses) return;

  if (!_sc->w) {
    cerr << "Classifier (" << _fileName << ") untrained." << endl;
    return;
  }

  register int i, j;
  register MarkExampleSet *exs;
  MarkExample    ex;
  unsigned int nexamples;

  for (i = 0; i < nclasses; ++i)
  {
    exs = catalog->example_set(classNames[i]);
    if (!exs) continue;

    //  classify all the examples in the set
    nexamples = exs->nexamples();

    for (j = 0; j < nexamples; ++j)
      if ( (exs->get_example(ex, j)) == j) {
        Vector fv = example_to_fv(ex);
        const char *className = classify_fv(fv);
        if(className)
          printf("%s\n", className);
```

```
      }
  }

} // classify_examples()


/*****************************************************************************/
/* classify_example()                                                        */
/*****************************************************************************/

const char *
MarkClassifier::classify_example(const MarkExample* ex_to_classify) const
{
  if(!ex_to_classify) return NULL;
  if (!_sc || !_sc->nclasses) return NULL;

  if (!_sc->w) {
    cerr << "Classifier (" << _fileName << ") untrained." << endl;
    return NULL;
  }

  Vector fv = example_to_fv(*ex_to_classify);
  const char *className = classify_fv(fv);
  if (!className) return NULL;

  //    CHECK IF THE RECOGNIZED CLASS IS DISABLED
  int ix = index(className);
  if (_classStatus[ix] == RECOG_DISABLED) return NULL;

  // char *name = new char [strlen(className) + 1];
  // strcpy(name, className);
  return className;

} // classify_example()


/*****************************************************************************/
/* classify_fv()                                                             */
/*****************************************************************************/

const char*
MarkClassifier::classify_fv(Vector fv) const
{
  double ap, dp;
  const char *className = classify_fv(fv, &ap, &dp);
  if (className) printf("%s, prob = %f, dist = %f\n", className, ap, dp);

  //    reject far-off or ambiguous examples
  if (dp <= MAHALANOBISDIST_THRESHOLD)
    return className;
  else
    return NULL;

} // classify_fv()


/*****************************************************************************/
/* classify_fv()                                                             */
/*****************************************************************************/

const char*
MarkClassifier::classify_fv(Vector fv, double *ap, double *dp) const
{
  if (!_sc || !_sc->nclasses) return NULL;

  sClassDope scd = sClassifyAD(_sc, fv, ap, dp);
  if (scd && scd->name)
    return scd->name;
  else
    return NULL;

} // classify_fv



/*****************************************************************************/
/* train()                                                                   */
/*****************************************************************************/

void
MarkClassifier::train()
{
  if(_sc && _sc->nclasses)
  {
    //  free existing "_sc->w, _sc->cnst and _sc->invavgcov"
    //  because they will be newly allocated in sDoneAdding() anyway.
/*
    if (_sc->cnst) FreeVector(_sc->cnst);
    if (_sc->invavgcov) FreeMatrix(_sc->invavgcov);
    for (int c = 0; c < _sc->nclasses; ++c)
      if(_sc->w && _sc->w[c]) FreeVector(_sc->w[c]);
    if (_sc->w) free(_sc->w);

*/
    sDoneAdding(_sc);
    _changed = 1;
  }

} //train()


/*****************************************************************************/
/* index()                                                                   */
/*****************************************************************************/

int
MarkClassifier::index (const char *className) const
{
  if (!_sc) return -1;

  int _nclasses = _sc->nclasses;
  if (!className || !_nclasses) {
    return -1;
  }

  register short i;

  for (i = 0; i < _nclasses; i++)
    if (!strcmp(_sc->classdope[i]->name, className)) return i;

  return -1;

} //index()


/*****************************************************************************/
/* example_to_fv()                                                           */
/*****************************************************************************/

Vector
MarkClassifier::example_to_fv (const MarkExample &ex) const
{
  register uint i;
  register uint npoints;
```

```
  register int x, y;
  register long time;
  register MarkPoint *mp;

  Vector fv;
  static FV fvInfo;
  if (!fvInfo) fvInfo = FvAlloc();

  FvInit (fvInfo);

  npoints = ex.npoints();

  for (i = 0; i < npoints; i++) {
    mp = ex.get_point(i);
    if (mp) {
      x = mp->x, y = mp->y, time = mp->time;
      FvAddPoint (fvInfo, x, y, time);
    }
  }

  fv = FvCalc(fvInfo);  // calculate the feature-vector

  // PrintVector(fv, ex.name());

  return fv;

} //example_to_fv()



/*****************************************************************************/
/* cleanup                                                                   */
/*****************************************************************************/

void
MarkClassifier::cleanup()
{
  delete _fileName; _fileName = NULL;

  if (_sc) {
    for (short i = 0; i < _sc->nclasses; ++i) {
      delete _classNames[i]; _classNames[i] = NULL;
      _classStatus[i] = RECOG_DISABLED;
      _repExamples[i].cleanup();
    }
    sFreeClassifier(_sc);
    _sc = sNewClassifier();
    _sc->cnst = NULL;
    _sc->invavgcov = NULL;
  }

  _changed = 0;

} //cleanup()


/*****************************************************************************/
/* class_names()                                                             */
/*****************************************************************************/

const char **
MarkClassifier::class_names() const
{
  return _classNames;
}

/*****************************************************************************/
```

```
/* set_file_name()                                                           */
/*****************************************************************************/

void
MarkClassifier::set_file_name(const char *new_name)
{
  if(!new_name) return;
  delete _fileName;
  _fileName = new char [strlen(new_name) + 1];
  strcpy(_fileName, new_name);
}



/*****************************************************************************/
/* toggle_recognition_status(index)                                          */
/*****************************************************************************/

void
MarkClassifier::toggle_recognition_status(int class_index)
{
  if (class_index < 0 || class_index >= _sc->nclasses)
    return;

  if (_classStatus[class_index] == RECOG_ENABLED)
    _classStatus[class_index] = RECOG_DISABLED;
  else
    _classStatus[class_index] = RECOG_ENABLED;

}


/*****************************************************************************/
/* toggle_recognition_status(class_name)                                     */
/*****************************************************************************/

void
MarkClassifier::toggle_recognition_status(const char *className)
{
  toggle_recognition_status(index(className));
}


/*****************************************************************************/
/* enabled(class_index)                                                      */
/*****************************************************************************/

RecogStatus
MarkClassifier::enabled(int class_index) const
{
  if (class_index < 0 || class_index >= _sc->nclasses)
    return RECOG_DISABLED;

  return _classStatus[class_index];

}


/*****************************************************************************/
/* enabled(class_index)                                                      */
/*****************************************************************************/

RecogStatus
MarkClassifier::enabled(const char *class_name) const
{
  return enabled(index(class_name));
```

```
}
```

```
/*****************************************************************************/
/* MarkEditor.C                                                              */
/*                                                                           */
/* methods for class MarkEditor                                              */
/*                                                                           */
/* Author:  Lalit K. Agarwal, Brown University                               */
/* Date  :  May 1994                                                         */
/*****************************************************************************/


#include "MarkEditor.H"

#include <X11/Intrinsic.h>      /* Intrinsics Definitions */
#include <X11/StringDefs.h>     /* Standard Name-String definitions */

#include <Xm/Xm.h>
#include <Xm/MainW.h>
#include <Xm/DrawingA.h>
#include <Xm/Label.h>
#include <Xm/PushBG.h>
#include <Xm/PushB.h>
#include <Xm/CascadeBG.h>
#include <Xm/CascadeB.h>
#include <Xm/RowColumn.h>
#include <Xm/SeparatoG.h>
#include <Xm/Separator.h>
#include <Xm/List.h>
#include <Xm/Form.h>
#include <Xm/Text.h>
#include <Xm/SelectioB.h>
#include <Xm/PanedW.h>
#include <Xm/DialogS.h>
#include <Xm/MessageB.h>
#include <Xm/ScrolledW.h>

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <iostream.h>

#define ASTERIX_TERMINATION " *"

XmStringCharSet charset = XmSTRING_DEFAULT_CHARSET;

#define NAME_START_X      10
#define NAME_START_Y      10
#define MAXLINE           100
#define KEYCODE_D         86

/*****************************************************************
*  F O R W A R D     D E C L A R A T I O N S   F O R
*             C A L L B A C K S
*****************************************************************/

/******* Callbacks for "File" menu *********/

//  CATALOG file menu

static void
handle_cat_open(Widget, XtPointer, void*);

static void
handle_cat_new(Widget, XtPointer, void*);

static void
handle_cat_save(Widget, XtPointer, void*);

static void
handle_cat_saveAll(Widget, XtPointer, void*);

void
handle_cat_save_list (Widget w, XtPointer client_data, void *cb);

static void
handle_cat_save_as(Widget, XtPointer, void*);

static void
handle_cat_close(Widget, XtPointer, void*);

void
handle_cat_close_list (Widget w, XtPointer client_data, void *cb);


//  RECOGNIZER  file menu

static void
handle_rec_new(Widget, XtPointer, void*);

static void
handle_rec_open(Widget, XtPointer, void*);

static void
handle_rec_save(Widget, XtPointer, void*);

static void
handle_rec_saveAll(Widget, XtPointer, void*);

void
handle_rec_save_list (Widget w, XtPointer client_data, void *cb);

static void
handle_rec_save_as(Widget, XtPointer, void*);

static void
handle_rec_close(Widget, XtPointer, void*);

void
handle_rec_close_list (Widget w, XtPointer client_data, void *cb);


static void
handle_quit(Widget, XtPointer, void*);

static void
quit_save_callback(Widget, XtPointer client_data, void *);

static void
quit_exit_callback(Widget, XtPointer client_data, void *);


/******* Callbacks for "Edit" menu *********/

//      CATALOG edit menu

static void
handle_cat_selectCatList(Widget, XtPointer, void*);

static void
handle_cat_showMarkExamples(Widget, XtPointer, void*);

static void
handle_cat_addMarkExample(Widget, XtPointer, void*);

static void
```

```
handle_cat_setRepExample(Widget, XtPointer, void*);

static void
handle_cat_removeClass(Widget, XtPointer, void*);

//      CLASSIFIER edit menu

static void
handle_rec_selectRecList(Widget, XtPointer, void*);

static void
handle_rec_setRepExample(Widget, XtPointer, void*);

static void
handle_rec_addMarks(Widget, XtPointer, void*);

static void
handle_rec_drawAddMarks(Widget, XtPointer, void*);

static void
handle_rec_removeClass(Widget, XtPointer, void*);

static void
handle_rec_train(Widget, XtPointer, void*);

static void
handle_rec_classifyExamples(Widget, XtPointer, void*);


/******* Callbacks for "Help" menu *********/

static void
handle_help(Widget, XtPointer, void*);


/******* Callback for Drawing Area *********/
static void
handle_da_callback(Widget, XtPointer, void*);


static void
handle_list (Widget, XtPointer, void*);


/******* Callbacks from PromptDialogs ******/

static void
open_catalog_ok_callback (Widget prompt_w, XtPointer client_data,
                          XmSelectionBoxCallbackStruct *cbs);

static void
new_catalog_ok_callback (Widget prompt_w, XtPointer client_data,
                          XmSelectionBoxCallbackStruct *cbs);

static void
save_catalog_ok_callback(Widget prompt_w, XtPointer client_data,
                         XmSelectionBoxCallbackStruct *cbs);

static void
open_classifier_ok_callback (Widget prompt_w, XtPointer client_data,
                             XmSelectionBoxCallbackStruct *cbs);
static void
new_classifier_ok_callback (Widget prompt_w, XtPointer client_data,
                             XmSelectionBoxCallbackStruct *cbs);

static void
save_classifier_ok_callback(Widget prompt_w, XtPointer client_data,
                            XmSelectionBoxCallbackStruct *cbs);

static void
prompt_get_className_ok_callback(Widget prompt_w, XtPointer client_data,
                                 XmSelectionBoxCallbackStruct *cbs);

static void
cat_selection_callback(Widget, XtPointer client_data , void *cb);

static void
rec_selection_callback(Widget, XtPointer client_data , void *cb);

static void
file_cancel_callback(Widget w, XtPointer, void *);

static void
cat_addNewClass_callback(Widget, XtPointer client_data , void *cbs);

static void
rec_addNewClass_callback(Widget, XtPointer client_data , void *cbs);


//      MISC. FUNCTIONS

Widget
get_top_shell(Widget w);

void
destroy_shell(Widget w, void *shell, void *);


/*****************************************************************
*                INITIALIZATION OF
*                PULL-DOWN MENUs' Data Structures
*
*****************************************************************/

/* initialize file-menu-items array */

static MarkMenuItem file_menu_items[] = {
  { "Catalog",      '\0',   "",           "",          &xmLabelWidgetClass,      NULL},
  { "Sep",          '\0',   "",           "",          &xmSeparatorGadgetClass,  NULL},
  { "Open",         'O',    "Ctrl<Key>O", "Ctrl+O",    &xmPushButtonGadgetClass, handle_ca
t_open },
  { "New",          'N',    "",           "",          &xmPushButtonGadgetClass, handle_ca
t_new },
  { "Save",         'S',    "Ctrl<Key>S", "Ctrl+S",    &xmPushButtonGadgetClass, handle_ca
t_save },
  { "Save All",     '\0',   "",           "",          &xmPushButtonGadgetClass, handle_ca
t_saveAll },
  { "Save As",      '\0',   "",           "",          &xmPushButtonGadgetClass, handle_ca
t_save_as },
  { "Close",        'C',    "Ctrl<Key>C", "Ctrl+C",    &xmPushButtonGadgetClass, handle_ca
t_close },
  { "Sep",          '\0',   "",           "",          &xmSeparatorGadgetClass,  NULL},
  { "Classifier",   '\0',   "",           "",          &xmLabelWidgetClass,      NULL},
  { "Sep",          '\0',   "",           "",          &xmSeparatorGadgetClass,  NULL},
  { "New ",         '\0',   "Ctrl<Key>N", "Ctrl+N",    &xmPushButtonGadgetClass, handle_re
c_new },
  { "Open",         'L',    "Ctrl<Key>L", "Ctrl+L",    &xmPushButtonGadgetClass, handle_re
c_open },
  { "Save",         'W',    "Ctrl<Key>W", "Ctrl+W",    &xmPushButtonGadgetClass, handle_re
c_save },
  { "Save All",     '\0',   "",           "",          &xmPushButtonGadgetClass, handle_re
c_saveAll },
  { "Save As",      '\0',   "",           "",          &xmPushButtonGadgetClass, handle_re
```

```
c_save_as ),
  { "Close",       'D',   "Ctrl<Key>D",  "Ctrl+D",    &xmPushButtonGadgetClass, handle_r
ec_close ),
  { "Sep",         '\0',  "",            "",          &xmSeparatorGadgetClass,  NULL),
  { "Quit",        'Q',   "Ctrl<Key>Q",  "Ctrl+Q",    &xmPushButtonGadgetClass, handle_q
uit ),
  NULL,
};

static MarkMenuItem edit_menu_items[] = {
  { "Catalog",            '\0', "",  "",          &xmLabelWidgetClass,      NULL),
  { "Sep",                '\0', "",  "",          &xmSeparatorGadgetClass,  NULL),
  { "Select Catalog",     '\0', "",  "",          &xmPushButtonGadgetClass, handle_cat_s
electCatList ),
  { "Show Mark Examples", '\0', "",  "",          &xmPushButtonGadgetClass, handle_cat_s
howMarkExamples ),
  { "Add Mark Example(s)",'\0', "",  "",          &xmPushButtonGadgetClass, handle_cat_a
ddMarkExample ),
  { "Set Rep. Example",   '\0', "",  "",          &xmPushButtonGadgetClass, handle_cat_s
etRepExample ),
  { "Remove Class",       '\0', "",  "",          &xmPushButtonGadgetClass, handle_cat_r
emoveClass ),
  { "Sep",                '\0', "",  "",          &xmSeparatorGadgetClass,  NULL),
  { "Classifier",         '\0', "",  "",          &xmLabelWidgetClass,      NULL),
  { "Sep",                '\0', "",  "",          &xmSeparatorGadgetClass,  NULL),
  { "Select Classifier",  '\0', "",  "",          &xmPushButtonGadgetClass, handle_rec_s
electRecList ),
  { "Set Rep. Example",   '\0', "",  "",          &xmPushButtonGadgetClass, handle_rec_s
etRepExample ),
  { "Add Class(es) from Catalog",  '\0', "",  "", &xmPushButtonGadgetClass, handle_rec_a
ddMarks),
  { "Add Mark Example (draw)",   '\0', "",  "",   &xmPushButtonGadgetClass, handle_rec_d
rawAddMarks),
//  { "Train",              '\0', "",  "",          &xmPushButtonGadgetClass, handle_rec
_train),
  { "Remove Class",       '\0', "",  "",          &xmPushButtonGadgetClass, handle_rec_r
emoveClass ),
  { "Classify Examples",  '\0', "",  "",          &xmPushButtonGadgetClass, handle_rec_c
lassifyExamples),
  NULL,
};

static MarkMenuItem help_items[] = {
  { "Tutorial", 'T', "<Key>H", "H",  &xmPushButtonGadgetClass, handle_help ),
  NULL,
};


/*****************************************************************/
/* MarkEditor                                                    */
/*****************************************************************/

MarkEditor::MarkEditor(Widget toplevel)
{
  Widget temp_w;

  toplevel_w = toplevel;

  main_w   = XtVaCreateManagedWidget("main_window",
                                     xmMainWindowWidgetClass,
                                     toplevel,
                                     NULL);


  //    CREAT PULL-DOWN MENUS
  create_menus();

  //    CREATE WORK AREA (also creates the message area)
  create_workArea();

  // catalog = &_catalogs[0];
  // classifier = &_classifiers[0];

  catalog = NULL;
  classifier = NULL;

  _ncatalogs = 0;
  _nclassifiers = 0;

  _cat_selection.item = NULL;
  for (register int i = 0; i < UI_MAX_SELECTED_ITEMS; ++i)
    _cat_selection.selected_items[i] = NULL;
  _cat_selection.selected_item_count = 0;
  _cat_selection.selected_item_positions = NULL;

  _rec_selection.item = NULL;
  for (i = 0; i < UI_MAX_SELECTED_ITEMS; ++i)
    _rec_selection.selected_items[i] = NULL;
  _rec_selection.selected_item_count = 0;
  _rec_selection.selected_item_positions = NULL;

  edit_mode = NONE;
  cur_exampleSet = NULL;
  cur_example = NULL;

} // MarkEditor()



/*****************************************************************/
/* build_pdm()                                                   */
/*****************************************************************/

Widget
MarkEditor::build_pdm(char *menu_title, char menu_mnemonic,
          MarkMenuItem *menu_items)
{
  //  create a pull-down menu
  Widget pull_down = XmCreatePulldownMenu(menubar_w, "pdm", NULL, 0);

  XmString title_str = XmStringCreateSimple(menu_title);

  //  create cascade-button as the menubar title
  //  the pull-down-menu (menu-items) is attached to it.
  Widget cascade_btn = XtVaCreateManagedWidget(menu_title,
                                     xmCascadeButtonWidgetClass, menubar_w,
                                     XmNsubMenuId, pull_down,
                                     XmNlabelString, title_str,
                                     XmNmnemonic,    menu_mnemonic,
                                     NULL);
  XmStringFree(title_str);

  //   now add the menu items to the cascade button

  for(int i = 0; menu_items[i].label_str != NULL; i++){
    Widget menu_item_widget = XtVaCreateManagedWidget(
                                menu_items[i].label_str,
                                *menu_items[i].class_ptr, pull_down,
                                NULL);
    XtVaSetValues(menu_item_widget, XmNmnemonic, menu_items[i].mnemonic, NULL);

    //if item has an accelerator
    if(menu_items[i].accelerator){
      XmString accel_str = XmStringCreateSimple(menu_items[i].accel_text);
      XtVaSetValues(menu_item_widget,
```

```
                    XmNaccelerator, menu_items[i].accelerator,
                    XmNacceleratorText, accel_str,
                    NULL);
      XmStringFree(accel_str);
    }//end of if item has accelerator

    if(menu_items[i].callback)
      XtAddCallback(menu_item_widget, XmNactivateCallback,
                    (XtCallbackProc) menu_items[i].callback, (XtPointer)this);

  }//end of building pull-down menu

  return cascade_btn;

} // build_pdm()


/*****************************************************************/
/* create_menus()                                                */
/*****************************************************************/

void
MarkEditor::create_menus()
{
  Widget temp_widget;

  //create the menu bar
  menubar_w = XmCreateMenuBar(main_w, "menubar", NULL, 0);

  (void) build_pdm("File", 'F', file_menu_items);
  (void) build_pdm("Edit", 'E', edit_menu_items);
  temp_widget =  build_pdm("Help", 'H', help_items);
  XtVaSetValues(menubar_w, XmNmenuHelpWidget, temp_widget, NULL);

  XtInstallAccelerators(main_w, menubar_w);
  XtManageChild(menubar_w);

} // create_menus()


/*****************************************************************/
/* create_workArea()                                             */
/*****************************************************************/

void
MarkEditor::create_workArea()
{
  //    CREATE WORK-AREA
  form_w = XtVaCreateWidget ("work_area", xmFormWidgetClass,
                             main_w, NULL);

  XtVaSetValues(main_w,
                XmNworkWindow, form_w,
                NULL);

  //    CREATE SCROLLED-LISTS
  create_lists();

  //    CREATE DRAWING-AREA
  create_drawingArea();

  //    CREATE MESSAGE-AREA
  create_messageArea();

  XtManageChild(form_w);

} // create_workArea()
```

```
/*****************************************************************/
/* create_lists()                                                */
/*****************************************************************/

void
MarkEditor::create_lists()
{

  pane_w = XtVaCreateWidget("pane_lists", xmPanedWindowWidgetClass, form_w,
                            XmNsashWidth, 1,
                            XmNsashHeight, 1,
                            NULL);
  XtVaSetValues(pane_w,
                XmNleftAttachment, XmATTACH_FORM,
                XmNtopAttachment, XmATTACH_FORM,
                NULL);

  XmString cat_label_str = XmStringCreateSimple("Catalog: ");
  cat_label = XtVaCreateManagedWidget("cat_list_label",
                                      xmLabelWidgetClass, pane_w,
                                      XmNlabelString, cat_label_str,
                                      // XmNleftAttachment, XmATTACH_FORM,
                                      // XmNtopAttachment, XmATTACH_FORM,
                                      NULL);
  XmStringFree(cat_label_str);

  cat_list_w = XmCreateScrolledList(pane_w, "list_catalog", NULL, 0);

  XtVaSetValues (cat_list_w,
                 XmNvisibleItemCount, 10,
                 XmNselectionPolicy, XmEXTENDED_SELECT,
                 XmNscrollBarDisplayPolicy, XmSTATIC,
                 NULL);

  XtAddCallback(cat_list_w, XmNdefaultActionCallback,
                handle_list, (XtPointer)this);
  XtAddCallback(cat_list_w, XmNextendedSelectionCallback,
                handle_list, (XtPointer)this);

  XmString rec_label_str = XmStringCreateSimple("Classifier: ");
  rec_label = XtVaCreateManagedWidget("cat_list_label",
                                      xmLabelWidgetClass, pane_w,
                                      XmNlabelString, rec_label_str,
                                      NULL);
  XmStringFree(rec_label_str);

  rec_list_w = XmCreateScrolledList(pane_w, "list_classifier", NULL, 0);

  XtVaSetValues (rec_list_w,
                 XmNvisibleItemCount, 10,
                 XmNselectionPolicy, XmEXTENDED_SELECT,
                 XmNscrollBarDisplayPolicy, XmSTATIC,
                 NULL);

  XtAddCallback(rec_list_w, XmNdefaultActionCallback,
                handle_list, (XtPointer)this);
  XtAddCallback(rec_list_w, XmNextendedSelectionCallback,
                handle_list, (XtPointer)this);

  XtManageChild(cat_list_w);
  XtManageChild(rec_list_w);
  XtManageChild(pane_w);

} //create_lists()
```

```
/*****************************************************************/
/* create_drawingArea()                                          */
/*****************************************************************/

void
MarkEditor::create_drawingArea()
{
  String da_translations =
      "<KeyDown>:     DrawingAreaInput() ManagerGadgetKeyInput() \n\
       <KeyUp>:       DrawingAreaInput() ManagerGadgetKeyInput() \n\
       <Btn1Up>:      DrawingAreaInput() ManagerGadgetArm() \n\
       <Btn1Down>:    DrawingAreaInput() ManagerGadgetArm() \n\
       <Btn1Motion>: DrawingAreaInput() ManagerGadgetButtonMotion()";

  Widget scroll_w;

//   scroll_w = XtVaCreateManagedWidget("scrolled_win",
//                                      xmScrolledWindowWidgetClass, form_w,
//                                      XmNscrollingPolicy, XmAUTOMATIC,
//                                      XmNleftAttachment, XmATTACH_WIDGET,
//                                      XmNleftWidget, pane_w,
//                                      XmNtopAttachment, XmATTACH_FORM,
//                                      XmNrightAttachment, XmATTACH_FORM,
//                                      NULL);

  da_w = XtVaCreateWidget ("drawing_area",
                           xmDrawingAreaWidgetClass, form_w,
                           XmNtranslations,
                           XtParseTranslationTable(da_translations),
                           XmNleftAttachment, XmATTACH_WIDGET,
                           XmNleftWidget, pane_w,
                           XmNtopAttachment, XmATTACH_FORM,
                           XmNrightAttachment, XmATTACH_FORM,
//                         XmNbottomAttachment, XmATTACH_FORM,
                           NULL);

  XtAddCallback(da_w, XmNinputCallback, handle_da_callback, (XtPointer)this);
  XtAddCallback(da_w, XmNexposeCallback, handle_da_callback, (XtPointer)this);
  XtAddCallback(da_w, XmNresizeCallback, handle_da_callback, (XtPointer)this);

  XtVaGetValues(da_w,
                XmNwidth, &da_width,
                XmNheight, &da_height,
                NULL);

  da_display = XtDisplay(da_w);
  da_screen  = XtScreen(da_w);

  //    initialize the data members da_fg, da_bg with the fallback values
  XtVaGetValues(da_w,
                XmNforeground, &da_fg,
                XmNbackground, &da_bg,
                NULL);
  //    fg = black, bg = white

  //    initialize the graphic context
  da_gc = XCreateGC(da_display, RootWindowOfScreen(da_screen), 0, NULL);
  XtVaSetValues(da_w, XmNuserData, da_gc, NULL);

  //    set foreground to bg color to paint clear the pixmap
  XSetForeground(da_display, da_gc, da_bg );

  //    create a pixmap the same size as the drawing area.
  da_pixmap = XCreatePixmap(da_display, RootWindowOfScreen(da_screen),
                            da_width, da_height,
                            DefaultDepthOfScreen(da_screen) );
  //    clear the pixmap with bg color
  XFillRectangle(da_display, da_pixmap, da_gc, 0, 0, da_width, da_height);

  //    change foreground back to actual for drawing
  XSetForeground(da_display, da_gc, da_fg );

  //    setting up the font
  font_name = "-adobe-times-bold-r-normal--14-140-75-75-p-77-iso8859-1";
  font_struct = XLoadQueryFont(da_display, font_name);
  if(font_struct == NULL)
    printf("font_struct = null\n");

  da_win = XtWindow(da_w);

  XtManageChild(da_w);

} // create_drawingArea()


/*****************************************************************/
/* create_messageArea()                                          */
/*****************************************************************/

void
MarkEditor::create_messageArea()
{
  Arg   args[9];

  XtSetArg(args[0], XmNrows,                5);
  XtSetArg(args[1], XmNcolumns,             80);
  XtSetArg(args[2], XmNeditable,            False);
  XtSetArg(args[3], XmNwordWrap,            True);
  XtSetArg(args[4], XmNscrollHorizontal,    False);
  XtSetArg(args[5], XmNblinkRate,           0);
  XtSetArg(args[6], XmNeditMode,            XmMULTI_LINE_EDIT);
  XtSetArg(args[7], XmNautoShowCursorPosition, True);
  XtSetArg(args[8], XmNcursorPositionVisible, False);

  message_w = XmCreateScrolledText(form_w, "message_area", args, 9);
  XtVaSetValues(XtParent(message_w),
                XmNleftAttachment, XmATTACH_FORM,
                XmNtopAttachment, XmATTACH_WIDGET,
                XmNtopWidget,   da_w,
                XmNrightAttachment, XmATTACH_FORM,
                XmNbottomAttachment, XmATTACH_FORM,
                NULL);

  XtManageChild(message_w);

} // create_messageArea()


/*****************************************************************
* HANDLE : handle_cat_open
*
* callback to open a catalog
*****************************************************************/

static void
handle_cat_open(Widget, XtPointer client_data , void*)
{
  ((MarkEditor *)client_data)->prompt_open_new('c', 'o');
}

/*****************************************************************
* HANDLE : handle_cat_new
```

```
*
* callback to create a new catalog
*****************************************************************/

static void
handle_cat_new(Widget, XtPointer client_data , void*)
{
  ((MarkEditor *)client_data)->prompt_open_new('c', 'n');
}


/*****************************************************************
* HANDLE : handle_rec_new
*
* callback to create a new recognizer(classifier)
*****************************************************************/

static void
handle_rec_new(Widget, XtPointer client_data , void*)
{
  ((MarkEditor *)client_data)->prompt_open_new('r', 'n');
}



/*****************************************************************
* HANDLE : handle_rec_open
*
* callback to open an existing recognizer(classifier)
*****************************************************************/

static void
handle_rec_open(Widget, XtPointer client_data , void*)
{
  ((MarkEditor *)client_data)->prompt_open_new('r', 'o');
}


/*****************************************************************/
/* prompt_open_new()                                             */
/*****************************************************************/

void
MarkEditor::prompt_open_new(char doc_type, char open_or_new)
{

   XmString sel_str;
   if (doc_type == 'c') {
     if (_ncatalogs == UI_MAX_CATALOGS) {
       print_message("Cann't open any more catalogs. ", 'n');
       print_message("Close one before opening another.", 'c');
       return;
     }
   }
   else if (doc_type == 'r') {
     if (_nclassifiers == UI_MAX_CLASSIFIERS) {
       print_message("Cann't open any more classifiers. ", 'n');
       print_message("Close one before opening another.", 'c');
       return;
     }
   }

   //   create the prompt dialog box
   prompt_dialog = XmCreatePromptDialog(toplevel_w,
                                        "prompt_open_new",
                                        NULL, 0);
   if (doc_type == 'c') {
     if (open_or_new == 'o') {
       sel_str = XmStringCreateSimple("Enter CATALOG file to open:");
       XtAddCallback(prompt_dialog, XmNokCallback,
                     (XtCallbackProc)open_catalog_ok_callback,
                     (XtPointer)this);
     }
     else if (open_or_new == 'n') {
       sel_str = XmStringCreateSimple("Enter name of NEW catalog:");
       XtAddCallback(prompt_dialog, XmNokCallback,
                     (XtCallbackProc)new_catalog_ok_callback,
                     (XtPointer)this);
     }
   }
   else if (doc_type == 'r') {
     if (open_or_new == 'o') {
       sel_str = XmStringCreateSimple("Enter CLASSIFIER file to open:");
       XtAddCallback(prompt_dialog, XmNokCallback,
                     (XtCallbackProc)open_classifier_ok_callback,
                     (XtPointer)this);
     }
     else if (open_or_new == 'n') {
       sel_str = XmStringCreateSimple("Enter name of NEW classifier:");
       XtAddCallback(prompt_dialog, XmNokCallback,
                     (XtCallbackProc)new_classifier_ok_callback,
                     (XtPointer)this);
     }
   }

   XtVaSetValues(prompt_dialog,
                 XmNselectionLabelString, sel_str,
                 XmNautoUnmanage,         False,
                 NULL);

   XmStringFree(sel_str);

   XtAddCallback(prompt_dialog, XmNcancelCallback,
                 (XtCallbackProc)file_cancel_callback, NULL);

   XtUnmanageChild(XmSelectionBoxGetChild(prompt_dialog,
                                          XmDIALOG_HELP_BUTTON));
   XtManageChild(prompt_dialog);
   XtPopup(XtParent(prompt_dialog), XtGrabNone);

} // prompt_open_new()



/*****************************************************************
* HANDLE : open_catalog_ok_callback
*
* callback from ok button of prompt dialog to get catalog name
*****************************************************************/

static void
open_catalog_ok_callback (Widget prompt_w, XtPointer client_data,
                          XmSelectionBoxCallbackStruct *cbs)
{
  char  *file_name;

  if( !XmStringGetLtoR(cbs->value, charset, &file_name) ){
    printf("Internal error\n");
    XtDestroyWidget(prompt_w);          //prompt_w no longer needed
    return;
  }

  XtDestroyWidget(prompt_w);    //prompt_w no longer needed
```

```
  ((MarkEditor *)client_data)->read_catalog(file_name, 'o');

} //open_catalog_ok_callback()


/*****************************************************************
* HANDLE : new_catalog_ok_callback
*
* callback from ok button of prompt dialog to get catalog name
*****************************************************************/

static void
new_catalog_ok_callback (Widget prompt_w, XtPointer client_data,
                         XmSelectionBoxCallbackStruct *cbs)
{
  char  *file_name;

  if( !XmStringGetLtoR(cbs->value, charset, &file_name) ){
    printf("Internal error\n");
    XtDestroyWidget(prompt_w);          //prompt_w no longer needed
    return;
  }

  XtDestroyWidget(prompt_w);    //prompt_w no longer needed

  ((MarkEditor *)client_data)->read_catalog(file_name, 'n');

} //new_catalog_ok_callback()


/*****************************************************************
* HANDLE : read_catalog()
*****************************************************************/

void
MarkEditor::read_catalog(char *file_name, char open_or_new)
{
  if (!file_name) return;

  MarkCatalog *new_slot = NULL;

  //    search for the first empty catalog slot
  for (int i = 0; i < UI_MAX_CATALOGS; ++i) {
    if (_catalogs[i].catalog_file() == NULL) {
      new_slot = &_catalogs[i];
      break;
    }
  }

  if (i == UI_MAX_CATALOGS) {
    printf("MarkEditor::read_catalog():  something wierd\n");
    return;
  }

  //    do anything only when read is successful
  if (open_or_new == 'o') {
    if (new_slot->read_catalog(file_name)){
      _ncatalogs++;
      catalog = new_slot;
      show_list('c');
    }
  }
  else if (open_or_new == 'n') {
      _ncatalogs++;
      catalog = new_slot;
      catalog->set_file_name(file_name);
      show_list('c');
    }

} // read_catalog()


/*****************************************************************
* HANDLE : open_classifier_ok_callback
*
* callback from ok button of prompt dialog
*****************************************************************/

static void
open_classifier_ok_callback (Widget prompt_w, XtPointer client_data,
                             XmSelectionBoxCallbackStruct *cbs)
{
  char  *file_name;

  if( !XmStringGetLtoR(cbs->value, charset, &file_name) ){
    printf("Internal error\n");
    XtDestroyWidget(prompt_w);          //prompt_w no longer needed
    return;
  }

  XtDestroyWidget(prompt_w);    //prompt_w no longer needed


  ((MarkEditor *)client_data)->read_classifier(file_name, 'o');

} //open_classifier_ok_callback()


/*****************************************************************
* HANDLE : new_classifier_ok_callback
*
* callback from ok button of prompt dialog
*****************************************************************/

static void
new_classifier_ok_callback (Widget prompt_w, XtPointer client_data,
                            XmSelectionBoxCallbackStruct *cbs)
{
  char  *file_name;

  if( !XmStringGetLtoR(cbs->value, charset, &file_name) ){
    printf("Internal error\n");
    XtDestroyWidget(prompt_w);          //prompt_w no longer needed
    return;
  }

  XtDestroyWidget(prompt_w);    //prompt_w no longer needed


  ((MarkEditor *)client_data)->read_classifier(file_name, 'n');

} // new_classifier_ok_callback()



/*****************************************************************
* read_classifier()
*****************************************************************/

void
MarkEditor::read_classifier(char *file_name, char open_or_new)
{
  if (!file_name) return;
```

```
  MarkClassifier *new_slot = NULL;

  //    search for the first empty classifier slot
  for (int i = 0; i < UI_MAX_CLASSIFIERS; ++i) {
    if (_classifiers[i].file_name() == NULL) {
      new_slot = &_classifiers[i];      // change current classifier
      break;
    }
  }

  if (i == UI_MAX_CLASSIFIERS) {
    printf("MarkEditor::read_classifier():  something wierd\n");
    return;
  }

  if (open_or_new == 'o') {
    if (new_slot->read_classifier(file_name)) {
      _nclassifiers++;
      classifier = new_slot;
      show_list('r');
    }
  }
  else if (open_or_new == 'n') {
    _nclassifiers++;
    classifier = new_slot;
    classifier->set_file_name(file_name);
    show_list('r');
    // TODO: set classifier file name
  }

} // read_classifier()



/*****************************************************************
* HANDLE : handle_cat_save
*
* callback to save catalog
*****************************************************************/

static void
handle_cat_save(Widget, XtPointer client_data , void*)
{
  ((MarkEditor *)client_data)->prompt_save_close('c', 's');
}


/*****************************************************************
* HANDLE : handle_cat_close
*
* callback to close catalogs
*****************************************************************/

static void
handle_cat_close(Widget, XtPointer client_data , void*)
{
  ((MarkEditor *)client_data)->prompt_save_close('c', 'c');
}


/*****************************************************************
* HANDLE : handle_rec_save
*
* callback to save recognizers
*****************************************************************/

static void
handle_rec_save(Widget, XtPointer client_data , void*)
{
  ((MarkEditor *)client_data)->prompt_save_close('r', 's');
}

/*****************************************************************
* HANDLE : handle_rec_close
*
* callback to close recognizers
*****************************************************************/

static void
handle_rec_close(Widget, XtPointer client_data , void*)
{
  ((MarkEditor *)client_data)->prompt_save_close('r', 'c');
}


/*****************************************************************/
/* prompt_save_close()                                           */
/*****************************************************************/

void
MarkEditor::prompt_save_close(char list_type, char save_close_mode)
{

  Widget list_label_w, cancel_button_w, warning_label_w;
  XmString label_str, warning_str;
  Boolean changed_items = False;
  int i;

  if (list_type == 'c' && !_ncatalogs) {
    if (save_close_mode == 's')
      print_message("No Catalogs to Save.", 'n');
    else if (save_close_mode == 'c')
      print_message("No Catalogs to Close.", 'n');
    return;
  }

  if (list_type == 'r' && !_nclassifiers) {
    if (save_close_mode == 's')
      print_message("No Classifiers to Save.", 'n');
    else if (save_close_mode == 'c')
      print_message("No Classifiers to Close.", 'n');
    return;
  }

  unsigned short xpos, ypos;
  XtVaGetValues(toplevel_w, XmNx, &xpos, XmNy, &ypos, NULL);

  shell_w = XtVaCreatePopupShell("prompt_save_close", xmDialogShellWidgetClass,
                                 get_top_shell(main_w),
                                 XmNdeleteResponse, XmDESTROY,
                                 XmNx, xpos,
                                 XmNy, ypos,
                                 NULL);

  pane_w = XtVaCreateWidget("pane_save_close", xmPanedWindowWidgetClass, shell_w,
                            XmNsashWidth, 1,
                            XmNsashHeight, 1,
                            NULL);

  list_label_w = XtVaCreateManagedWidget("save_close_list_label",
                                         xmLabelWidgetClass, pane_w,
                                         NULL);
```

```
  file_list_w = XmCreateScrolledList(pane_w, "list_save_rec", NULL, 0);

  XtVaSetValues (file_list_w,
                 XmNvisibleItemCount, 10,
                 XmNselectionPolicy, XmEXTENDED_SELECT,
                 XmNscrollBarDisplayPolicy, XmSTATIC,
                 NULL);

  if (list_type == 'c') {
    if (save_close_mode == 's') {
      label_str = XmStringCreateSimple("Catalogs needing SAVE: Press <RETURN> when done"
);
      XtAddCallback(file_list_w, XmNdefaultActionCallback,
                    handle_cat_save_list, (XtPointer)this);
    }
    else if (save_close_mode == 'c') {
      label_str = XmStringCreateSimple("Catalogs to CLOSE: Press <RETURN> when done");
      XtAddCallback(file_list_w, XmNdefaultActionCallback,
                    handle_cat_close_list, (XtPointer)this);
    }

    //  populate list with all the open catalogs during close
    //   and only the changed catalogs during a save

    for (i = 0; i < UI_MAX_CATALOGS; ++i)
    {
      const char *fileName = _catalogs[i].catalog_file();
      if (fileName) {
        char *newName = new char [strlen(fileName) + 3];
        strcpy(newName, fileName);

        //      only changed items go on the save-list
        if (save_close_mode == 's') {
          if (_catalogs[i].changed()) {
            changed_items = True;
            XmListAddItemUnselected(file_list_w,
                                    XmStringCreateSimple(newName), 0);
          }
        }

        //      all opened files go on the close-list
        else if (save_close_mode == 'c') {
          if (_catalogs[i].changed()) {
            changed_items = True;
            strcat(newName, ASTERIX_TERMINATION);
          }
          XmListAddItemUnselected(file_list_w,
                                  XmStringCreateSimple(newName), 0);
        }
        delete newName;

      } // if fileName
    } // for
  } // if list_type == c

  else if (list_type == 'r') {
    if (save_close_mode == 's') {
      label_str = XmStringCreateSimple("Classifiers to SAVE: Press <RETURN> when done");
      XtAddCallback(file_list_w, XmNdefaultActionCallback,
                    handle_rec_save_list, (XtPointer)this);
    }
    else if (save_close_mode == 'c') {
      label_str = XmStringCreateSimple("Classifiers to CLOSE: Press <RETURN> when done")
;
      XtAddCallback(file_list_w, XmNdefaultActionCallback,
                    handle_rec_close_list, (XtPointer)this);
    }

    //  populate list with all the open classifiers during a close
    //   and only the changed catalogs during a save

    for (i = 0; i < UI_MAX_CLASSIFIERS; ++i)
    {
      const char *fileName = _classifiers[i].file_name();
      if (fileName) {
        char *newName = new char [strlen(fileName) + 3];
        strcpy(newName, fileName);

        //      only changed items go on the save-list
        if (save_close_mode == 's') {
          if (_classifiers[i].changed()) {
            changed_items = True;
            XmListAddItemUnselected(file_list_w,
                                    XmStringCreateSimple(newName), 0);
          }
        }

        //      all valid names go on the close-list
        else if (save_close_mode == 'c') {
          if (_classifiers[i].changed()) {
            strcat(newName, ASTERIX_TERMINATION);
            changed_items = True;
          }
          XmListAddItemUnselected(file_list_w,
                                  XmStringCreateSimple(newName), 0);
        }
        delete newName;

      } // if fileName
    } // for
  } // else if list_type == r

  XtVaSetValues(list_label_w,
                XmNlabelString, label_str,
                NULL);

  XmStringFree(label_str);

  //    do not need save_close dialog if no items need saving
  if (save_close_mode == 's' && !changed_items)
  {
    print_message("No items need saving!", 'n');
    XtDestroyWidget(shell_w);
    return;
  }

  XtManageChild(file_list_w);

  //    only true if items changed during closing.
  if (save_close_mode == 'c' && changed_items)
  {
    warning_str = XmStringCreateSimple("Items followed by \"*\" need saving. Closing them
would discard their recent changes.");


    warning_label_w = XtVaCreateManagedWidget("save_close_warning_label",
                                              xmLabelWidgetClass, pane_w,
                                              XmNlabelString, warning_str,
                                              NULL);
    XmStringFree(warning_str);
  }

  form_w = XtVaCreateWidget("form_save_close", xmFormWidgetClass, pane_w,
                            XmNfractionBase, 3,
```

```
                        NULL);

  cancel_button_w = XtVaCreateManagedWidget("Cancel",
                                            xmPushButtonGadgetClass, form_w,
                                            XmNtopAttachment,    XmATTACH_FORM,
                                            XmNbottomAttachment, XmATTACH_FORM,
                                            XmNleftAttachment, XmATTACH_POSITION,
                                            XmNleftPosition,     1,
                                            XmNrightAttachment, XmATTACH_POSITION,
                                            XmNrightPosition,    2,
                                            XmNshowAsDefault,    True,
                                            XmNdefaultButtonShadowThickness, 1,
                                            NULL);
  XtAddCallback(cancel_button_w, XmNactivateCallback, destroy_shell, shell_w);

  XtManageChild(form_w);
  XtManageChild(pane_w);
  XtPopup(shell_w, XtGrabNone);

} // prompt_save_close()



/*****************************************************************
* HANDLE : handle_cat_save_list
*
* callback to list
*****************************************************************/

void
handle_cat_save_list (Widget w, XtPointer client_data, void *cb)
{
  XmListCallbackStruct *cbs = (XmListCallbackStruct *)cb;
  MarkEditor *ui = ((MarkEditor *)client_data);

  if (cbs->reason != XmCR_DEFAULT_ACTION) return;

  //    extract the list of items
  int nitems = cbs->selected_item_count;
  char *items[UI_MAX_SELECTED_ITEMS];
  char *item;

  for (int i = 0; i < nitems; ++i) {
    XmStringGetLtoR(cbs->selected_items[i], charset, &item);

    //  check for any asterix at the end
    char *starp = strstr(item, ASTERIX_TERMINATION);
    if (starp)
      *starp = '\0';

    items[i] = new char [strlen(item) + 1];
    strcpy(items[i], item);
    XtFree(item);
  }

  //    save the selected catalogs
  ui->save_catalogs(nitems, items);

  //    clean up
  for (i = 0; i < nitems; ++i)
    delete items[i];

  destroy_shell(w, (void *)XtParent(XtParent(XtParent(w))), NULL);

} // handle_cat_save_list()
```

```
/*****************************************************************
* HANDLE : handle_cat_close_list
*
* callback to list
*****************************************************************/

void
handle_cat_close_list (Widget w, XtPointer client_data, void *cb)
{
  XmListCallbackStruct *cbs = (XmListCallbackStruct *)cb;
  MarkEditor *ui = ((MarkEditor *)client_data);

  if (cbs->reason != XmCR_DEFAULT_ACTION) return;

  //    extract the list of items
  int nitems = cbs->selected_item_count;
  char *items[UI_MAX_SELECTED_ITEMS];
  char *item;

  for (int i = 0; i < nitems; ++i) {
    XmStringGetLtoR(cbs->selected_items[i], charset, &item);

    //  check for any asterix at the end
    char *starp = strstr(item, ASTERIX_TERMINATION);
    if (starp)
      *starp = '\0';

    items[i] = new char [strlen(item) + 1];
    strcpy(items[i], item);
    XtFree(item);
  }

  //    close the selected catalogs
  ui->close_catalogs(nitems, items);

  //    clean up
  for (i = 0; i < nitems; ++i)
    delete items[i];

  destroy_shell(w, (void *)XtParent(XtParent(XtParent(w))), NULL);

} // handle_cat_close_list()



/*****************************************************************
* save_catalogs()
*****************************************************************/

void
MarkEditor::save_catalogs(int nitems, char **items)
{
  for (int i = 0; i < nitems; ++i) {
    int ix = catalog_index(items[i]); // see if there's a match
    if (ix >= 0)
      _catalogs[ix].write_catalog(items[i]);
  }

} // save_catalogs()


/*****************************************************************
* close_catalogs()
*****************************************************************/

void
MarkEditor::close_catalogs(int nitems, char **items)
```

```
{
   if (!nitems || !items) return;

   int i, j, ix;
   int changed = 0;
   int *changed_item_positions = new int[nitems];
   MarkCatalog *close_cat;

   //   find out which ones have changed since last save
   for (i = 0, j = 0; i < nitems; ++i) {
     ix = catalog_index(items[i]);
     if (ix < 0) continue;
     close_cat = &_catalogs[ix];
     if (close_cat->changed()) {
       changed_item_positions[j++] = ix;
       changed = 1;
     }
   }

   //   now we can clean them up anyway
   for (i = 0; i < nitems; ++i) {
     ix = catalog_index(items[i]);
     if (ix < 0) continue;
     close_cat = &_catalogs[ix];
     close_cat->cleanup();
     _ncatalogs--;
     if (close_cat == catalog) {
       XmListDeleteAllItems(rec_list_w);   // TODO: BUG, crashes here
       clear_selection('c');
       clear_da();
       catalog = NULL;
       XmString list_label = XmStringCreateSimple("Catalog:");
       XtVaSetValues(rec_label, XmNlabelString, list_label, NULL);
       XmStringFree(list_label);
     }
     if (!_ncatalogs) {
       catalog = NULL;
       XmString list_label = XmStringCreateSimple("Catalog:");
       XtVaSetValues(rec_label, XmNlabelString, list_label, NULL);
       XmStringFree(list_label);
       break;
     }
   }

   delete changed_item_positions;

} // close_catalogs()


/*****************************************************************
* HANDLE : handle_rec_save_list
*
* callback to list
*****************************************************************/

void
handle_rec_save_list (Widget w, XtPointer client_data, void *cb)
{
  XmListCallbackStruct *cbs = (XmListCallbackStruct *)cb;
  MarkEditor *ui = ((MarkEditor *)client_data);

  if (cbs->reason != XmCR_DEFAULT_ACTION) return;

  //    extract the list of items
  int nitems = cbs->selected_item_count;
  char *items[UI_MAX_SELECTED_ITEMS];
  char *item;

  for (int i = 0; i < nitems; ++i) {
    XmStringGetLtoR(cbs->selected_items[i], charset, &item);

    //  check for any asterix at the end
    char *starp = strstr(item, ASTERIX_TERMINATION);
    if (starp)
      *starp = '\0';

    items[i] = new char [strlen(item) + 1];
    strcpy(items[i], item);
    XtFree(item);
  }

  //    save the selected classifiers
  ui->save_classifiers(nitems, items);

  //    clean up
  for (i = 0; i < nitems; ++i)
    delete items[i];

  destroy_shell(w, (void *)XtParent(XtParent(XtParent(w))), NULL);

} // handle_rec_save_list()



/*****************************************************************
* HANDLE : handle_rec_close_list
*
* callback to list
*****************************************************************/

void
handle_rec_close_list (Widget w, XtPointer client_data, void *cb)
{
  XmListCallbackStruct *cbs = (XmListCallbackStruct *)cb;
  MarkEditor *ui = ((MarkEditor *)client_data);

  if (cbs->reason != XmCR_DEFAULT_ACTION) return;

  //    extract the list of items
  int nitems = cbs->selected_item_count;
  char *items[UI_MAX_SELECTED_ITEMS];
  char *item;

  for (int i = 0; i < nitems; ++i) {
    XmStringGetLtoR(cbs->selected_items[i], charset, &item);

    //  check for any asterix at the end
    char *starp = strstr(item, ASTERIX_TERMINATION);
    if (starp)
      *starp = '\0';
    items[i] = new char [strlen(item) + 1];
    strcpy(items[i], item);
    XtFree(item);
  }

  //    save the selected classifiers
  ui->close_classifiers(nitems, items);

  //    clean up
  for (i = 0; i < nitems; ++i)
    delete items[i];

  destroy_shell(w, (void *)XtParent(XtParent(XtParent(w))), NULL);
```

```
} // handle_rec_close_list()


/*****************************************************************
* save_classifiers()
*****************************************************************/

void
MarkEditor::save_classifiers(int nitems, char **items)
{
  for (int i = 0; i < nitems; ++i) {
    int ix = classifier_index(items[i]); // see if there's a match
    if (ix >= 0)
      _classifiers[ix].write_classifier(items[i]);
  }

} // save_classifiers()


/*****************************************************************
* close_classifiers()
*****************************************************************/

void
MarkEditor::close_classifiers(int nitems, char **items)
{
   if (!nitems || !items) return;

   int i, j, ix;
   int changed = 0;
   int *changed_item_positions = new int[nitems];
   MarkClassifier *close_rec;

   //   find out which ones have changed since last save
   for (i = 0, j = 0; i < nitems; ++i) {
     ix = classifier_index(items[i]);
     if (ix < 0) continue;
     close_rec = &_classifiers[ix];
     if (close_rec->changed()) {
       changed_item_positions[j++] = ix;
       changed = 1;
     }
   }

   //   now we can clean them up anyway
   for (i = 0; i < nitems; ++i) {
     ix = classifier_index(items[i]);
     if (ix < 0) continue;
     close_rec = &_classifiers[ix];
     close_rec->cleanup();
     _nclassifiers--;
     if (close_rec == classifier) {
       XmListDeleteAllItems(rec_list_w); // TODO: BUG, crashes here
       clear_selection('r');
       clear_da();
       classifier = NULL;
       XmString list_label = XmStringCreateSimple("Classifier:");
       XtVaSetValues(rec_label, XmNlabelString, list_label, NULL);
       XmStringFree(list_label);
     }
     if (!_nclassifiers) {
       classifier = NULL;
       XmString list_label = XmStringCreateSimple("Classifier:");
       XtVaSetValues(rec_label, XmNlabelString, list_label, NULL);
       XmStringFree(list_label);
       break;
     }
   }

   delete changed_item_positions;

} // close_classifiers()



/*****************************************************************
* HANDLE : handle_cat_save_as
*
* callback to save catalog as
*****************************************************************/

static void
handle_cat_save_as(Widget, XtPointer client_data , void*)
{
  ((MarkEditor *)client_data)->prompt_save_as('c');
}

/*****************************************************************
* HANDLE : handle_rec_save_as
*
* callback to save classifier as
*****************************************************************/

static void
handle_rec_save_as(Widget, XtPointer client_data , void*)
{
  ((MarkEditor *)client_data)->prompt_save_as('r');
}


/*****************************************************************
* prompt_save_as()
*****************************************************************/

void
MarkEditor::prompt_save_as(char doc_type)
{
    XmString sel_str;
    XmString prompt_str;

    if ((doc_type == 'c' && !catalog) || (doc_type == 'r' && !classifier))
      return;

    //create the prompt dialog box
    prompt_dialog = XmCreatePromptDialog(toplevel_w, "save_as_prompt",
                                         NULL, 0);
    if (doc_type == 'c') {
      sel_str = XmStringCreateSimple("Enter Catalog Name to save in:");
      prompt_str = XmStringCreateSimple((char *)catalog->catalog_file());
      XtAddCallback(prompt_dialog, XmNokCallback, (XtCallbackProc)save_catalog_ok_callbac
k, (XtPointer)this);
    }
    else if (doc_type == 'r') {
      sel_str = XmStringCreateSimple("Enter Classifier Name to save in:");
      prompt_str = XmStringCreateSimple((char *)classifier->file_name());
      XtAddCallback(prompt_dialog, XmNokCallback, (XtCallbackProc)save_classifier_ok_call
back, (XtPointer)this);
    }
    else
      return;

    XtVaSetValues(prompt_dialog,
                  XmNselectionLabelString, sel_str,
```

```
                XmNpromptString,        prompt_str,
                XmNautoUnmanage,        False,
                NULL);
    XtVaSetValues(XtParent(prompt_dialog),
                XmNtitle, "Save as Prompt",
                NULL);

    XmStringFree(sel_str);
    XmStringFree(prompt_str);

    XtAddCallback(prompt_dialog, XmNcancelCallback,
                (XtCallbackProc)file_cancel_callback, NULL);

    XtUnmanageChild(XmSelectionBoxGetChild(prompt_dialog,
                                        XmDIALOG_HELP_BUTTON));
    XtManageChild(prompt_dialog);
    XtPopup(XtParent(prompt_dialog), XtGrabNone);

} // prompt_save_as()


/*****************************************************************
* save_catalog_ok_callback()
*****************************************************************/

static void
save_catalog_ok_callback(Widget prompt_w, XtPointer client_data,
                        XmSelectionBoxCallbackStruct *cbs)
{
  char  *file_name;

  if( !XmStringGetLtoR(cbs->value, charset, &file_name) ){
    printf("Internal error\n");
    XtDestroyWidget(prompt_w);          //prompt_w no longer needed
    return;
  }

  XtDestroyWidget(prompt_w);    //prompt_w no longer needed
  ((MarkEditor *)client_data)->write_catalog(file_name);

} //save_catalog_ok_callback()


/*****************************************************************
* save_classifier_ok_callback()
*****************************************************************/

static void
save_classifier_ok_callback(Widget prompt_w, XtPointer client_data,
                        XmSelectionBoxCallbackStruct *cbs)
{
  char  *file_name;

  if( !XmStringGetLtoR(cbs->value, charset, &file_name) ){
    printf("Internal error\n");
    XtDestroyWidget(prompt_w);          //prompt_w no longer needed
    return;
  }

  XtDestroyWidget(prompt_w);    //prompt_w no longer needed
  ((MarkEditor *)client_data)->write_classifier(file_name);

} //save_classifier_ok_callback()



/*****************************************************************
* HANDLE : handle_cat_saveAll
*
* callback to save all catalogs
*****************************************************************/

static void
handle_cat_saveAll(Widget, XtPointer client_data , void*)
{
  ((MarkEditor *)client_data)->saveAll('c');
}


/*****************************************************************
* HANDLE : handle_rec_saveAll
*
* callback to save all recognizers
*****************************************************************/

static void
handle_rec_saveAll(Widget, XtPointer client_data , void*)
{
  ((MarkEditor *)client_data)->saveAll('r');
}


/*****************************************************************/
/* saveAll()                                                     */
/*****************************************************************/

void
MarkEditor::saveAll(char doc_type)
{
  int i;

  if (doc_type == 'c' && _ncatalogs)
  {
    for (i = 0; i < UI_MAX_CATALOGS; ++i) {
      if(_catalogs[i].changed()) {
        _catalogs[i].write_catalog();
      }
    }
  }

  if (doc_type == 'r' && _nclassifiers)
  {
    for (i = 0; i < UI_MAX_CLASSIFIERS; ++i) {
      if(_classifiers[i].changed()) {
        _classifiers[i].write_classifier();
      }
    }
  }

} // saveAll()



/*****************************************************************
* HANDLE : handle_quit
*
* callback to quit MarkEditor
*****************************************************************/

static void
handle_quit(Widget, XtPointer client_data , void*)
{
  ((MarkEditor*)client_data)->prompt_quit();
}
```

```
/*****************************************************************
* quit
*
*****************************************************************/

void
MarkEditor::prompt_quit()
{
  //    find out which ones have changed since last save

  int i;
  int cats_changed = 0;
  int recs_changed = 0;

  for (i = 0 ; i < UI_MAX_CATALOGS; ++i) {
    if(_catalogs[i].changed()) {
      cats_changed = 1;
      break;
    }
  }

  for (i = 0; i < UI_MAX_CLASSIFIERS; ++i) {
    if(_classifiers[i].changed()) {
      recs_changed = 1;
      break;
    }
  }

  Widget dialog;
  XmString msg, yes, no, help;
  dialog = XmCreateQuestionDialog(toplevel_w, "question_quit", NULL, 0);
  no = XmStringCreateSimple("Cancel");

  if (cats_changed | recs_changed)
  {
    //  ask user if she wants to save changes

    msg = XmStringCreateSimple("Catalogs/Classifiers modified. Save and Quit?");
    yes = XmStringCreateSimple("Save all and Quit");
    help = XmStringCreateSimple("Quit");

    XtVaSetValues(dialog,
                  XmNmessageString,     msg,
                  XmNokLabelString,     yes,
                  XmNcancelLabelString, no,
                  XmNhelpLabelString,   help,
                  NULL);

    XtAddCallback(dialog, XmNokCallback, quit_save_callback, (XtPointer)this);
    XtAddCallback(dialog, XmNcancelCallback, file_cancel_callback, NULL);
    XtAddCallback(dialog, XmNhelpCallback, quit_exit_callback, NULL);

    XmStringFree(help);
  }
  else {
    msg = XmStringCreateSimple("Quit?");
    yes = XmStringCreateSimple("Yes");

     XtVaSetValues(dialog,
                  XmNmessageString,     msg,
                  XmNokLabelString,     yes,
                  XmNcancelLabelString, no,
                  NULL);

    XtAddCallback(dialog, XmNcancelCallback, file_cancel_callback, NULL);
    XtAddCallback(dialog, XmNokCallback, quit_exit_callback, NULL);
    XtUnmanageChild(XmMessageBoxGetChild(dialog, XmDIALOG_HELP_BUTTON));

  }

  XmStringFree(msg);
  XmStringFree(yes);
  XmStringFree(no);

  XtManageChild(dialog);
  XtPopup(XtParent(dialog), XtGrabNone);

} // prompt_quit()


/*****************************************************************
* HANDLE : quit_save_callback
*
*****************************************************************/

static void
quit_save_callback(Widget, XtPointer client_data, void *)
{
  ((MarkEditor*)client_data)->save_quit();
}

/*****************************************************************
* HANDLE : quit_exit_callback
*
*****************************************************************/

static void
quit_exit_callback(Widget, XtPointer client_data, void *)
{
  exit(1);
}

/*****************************************************************
* save_quit()
*
*****************************************************************/

void
MarkEditor::save_quit()
{
  int i;

  for (i = 0; i < UI_MAX_CATALOGS; ++i) {
    if(_catalogs[i].changed()) {
      _catalogs[i].write_catalog();
    }
  }

  for (i = 0; i < UI_MAX_CLASSIFIERS; ++i) {
    if(_classifiers[i].changed()) {
      _classifiers[i].write_classifier();
    }
  }

  exit(1);

} // save_quit()


/*****************************************************************
* HANDLE : handle_cat_selectCatList
*
* callback
```

```
*****************************************************************/

static void
handle_cat_selectCatList(Widget w, XtPointer client_data, void*)
{
  ((MarkEditor*)client_data)->prompt_show_selection('c');
}


/*****************************************************************
* HANDLE : handle_rec_selectRecList
*
* callback
*****************************************************************/

static void
handle_rec_selectRecList(Widget w, XtPointer client_data, void*)
{
  ((MarkEditor*)client_data)->prompt_show_selection('r');

}


/*****************************************************************/
/* prompt_show_selection()                                       */
/*****************************************************************/

void
MarkEditor::prompt_show_selection(char sel)
{
  XmString list_label, ok_label, *list_strs;
  register int i, j;
  int nitems;

  list_strs = (XmString *)XtMalloc(_ncatalogs * sizeof(XmString));

  if (sel == 'c') {
    nitems = _ncatalogs;
    if (!nitems) {
      print_message("No Catalogs to select from.", 'n');
      return;
    }
    list_label = XmStringCreateSimple("Currently Opened Catalogs:");

    for (i = j = 0; i < UI_MAX_CATALOGS; ++i)
      if(_catalogs[i].catalog_file())
        list_strs[j++] = XmStringCreateSimple((char *)_catalogs[i].catalog_file());

  }
  else if (sel == 'r') {
    nitems = _nclassifiers;
    if (!nitems) {
      print_message("No Classifiers to select from.", 'n');
      return;
    }
    list_label = XmStringCreateSimple("Currently Open Classifiers:");

    for (i = j = 0; i < UI_MAX_CLASSIFIERS; ++i)
      if(_classifiers[i].file_name())
        list_strs[j++] = XmStringCreateSimple((char *)_classifiers[i].file_name());
  }

  ok_label = XmStringCreateSimple("Select");

  selection_dialog = XmCreateSelectionDialog(toplevel_w, "prompt_selection", NULL, 0);
  XtVaSetValues(selection_dialog,
                XmNlistLabelString,     list_label,
                XmNlistItems,           list_strs,
                XmNlistItemCount,       nitems,
                XmNmustMatch,           True,
                XmNokLabelString,       ok_label,
                NULL);

  XtUnmanageChild(XmSelectionBoxGetChild(selection_dialog,
                                         XmDIALOG_HELP_BUTTON));
  XtUnmanageChild(XmSelectionBoxGetChild(selection_dialog,
                                         XmDIALOG_APPLY_BUTTON));


  XtAddCallback(selection_dialog, XmNcancelCallback, file_cancel_callback,
                NULL);

  if (sel == 'c') {
    XtAddCallback(selection_dialog, XmNokCallback, cat_selection_callback,
                  (XtPointer) this);
    XtAddCallback(selection_dialog, XmNnoMatchCallback, cat_selection_callback,
                  (XtPointer) this);
  }
  else if (sel == 'r') {
    XtAddCallback(selection_dialog, XmNokCallback, rec_selection_callback,
                  (XtPointer) this);
    XtAddCallback(selection_dialog, XmNnoMatchCallback, rec_selection_callback,
                  (XtPointer) this);
  }

  XmStringFree(list_label);
  XmStringFree(ok_label);

  while (--j >=0)
    XmStringFree(list_strs[j]);
  XtFree((char *) list_strs);
  XtManageChild(selection_dialog);

} // prompt_show_selection()



/*****************************************************************/
/* handle: rec_selection_callback ()                             */
/*****************************************************************/

static void
rec_selection_callback(Widget w, XtPointer client_data , void *cb)
{
  XmSelectionBoxCallbackStruct *cbs = (XmSelectionBoxCallbackStruct *)cb;
  char *selection;
  MarkEditor *ui = (MarkEditor *)client_data;

  XmStringGetLtoR(cbs->value, XmSTRING_DEFAULT_CHARSET, &selection);

  switch (cbs->reason) {
  case XmCR_OK:
    ui->show_selection('r', selection);
    break;
  case XmCR_APPLY:
    ui->close_selection('r', selection);
    break;
  case XmCR_NO_MATCH:
    ui->print_message("Classifier not found.\n", 'n');
    break;
  default:
    ui->print_message("Unknown Selection.\n", 'n');
    break;
  }
```

```
  XtDestroyWidget(w);

} // rec_selection_callback()


/*****************************************************************/
/* handle: cat_selection_callback ()                             */
/*****************************************************************/

static void
cat_selection_callback(Widget w, XtPointer client_data , void *cb)
{
  XmSelectionBoxCallbackStruct *cbs = (XmSelectionBoxCallbackStruct *)cb;
  char *selection;
  MarkEditor *ui = (MarkEditor *)client_data;

  XmStringGetLtoR(cbs->value, XmSTRING_DEFAULT_CHARSET, &selection);

  switch (cbs->reason) {
  case XmCR_OK:
    ui->show_selection('c', selection);
    break;
  case XmCR_APPLY:
    ui->close_selection('c', selection);
    break;
  case XmCR_NO_MATCH:
    ui->print_message("Catalog not found.\n", 'n');
    break;
  default:
    ui->print_message("Unknown Selection\n", 'n');
    break;
  }

  XtDestroyWidget(w);

} // cat_selection_callback()


/*****************************************************************/
/* show_selection()                                              */
/*****************************************************************/

void
MarkEditor::show_selection(char sel_type, char *sel_str)
{

  if (sel_type == 'c')
  {
    for (int i = 0; i < UI_MAX_CATALOGS; ++i) {
      if (_catalogs[i].catalog_file())
        if (!strcmp(sel_str, _catalogs[i].catalog_file()) ){
          catalog = &_catalogs[i];
          show_list('c');
        )
    }
  }
  else if (sel_type == 'r')
  {
    for (int i = 0; i < UI_MAX_CLASSIFIERS; ++i) {
      if (_classifiers[i].file_name())
          if (!strcmp(sel_str, _classifiers[i].file_name()) ){
            classifier = &_classifiers[i];
            show_list('r');
          )
    }
  )

} // show_selection()


/*****************************************************************/
/* close_selection()                                             */
/*****************************************************************/

void
MarkEditor::close_selection(char sel_type, char *sel_str)
{

  if (sel_type == 'c')
  {
    for (int i = 0; i < UI_MAX_CATALOGS; ++i) {
      if (_catalogs[i].catalog_file())
        if (!strcmp(sel_str, _catalogs[i].catalog_file()) ){
          MarkCatalog *close_catalog = &_catalogs[i];

          // TODO; ask for save before cleanup if catalog changed

          close_catalog->cleanup();
          _ncatalogs--;
          if (close_catalog == catalog) {
            XmListDeleteAllItems(cat_list_w);
            clear_selection('c');
            catalog = NULL;
          )
          if (!_ncatalogs)
            catalog = NULL;
        )
    }
  )
  else if (sel_type == 'r')
  {
    for (int i = 0; i < UI_MAX_CLASSIFIERS; ++i) {
      if (_classifiers[i].file_name())
        if (!strcmp(sel_str, _classifiers[i].file_name()) ){
          MarkClassifier *close_classifier = &_classifiers[i];

          // TODO; ask for save before cleanup if catalog changed

          close_classifier->cleanup();
          _nclassifiers--;
          if (close_classifier == classifier) {
            XmListDeleteAllItems(rec_list_w);
            clear_selection('r');
            classifier = NULL;
          )
          if (!_nclassifiers)
            classifier = NULL;
        )
    }
  )

} // close_selection()


/*****************************************************************/
/* show_list()                                                   */
/*****************************************************************/

void
MarkEditor::show_list(char listType)
{
  XmString list_label;
  Widget label_w;
```

```
  if (listType == 'c' || listType == 'r')
  {
    //  clear up the current list
    Widget list_w;
    char** class_names;
    unsigned int nclasses;

    if (listType == 'c') {
      list_w = cat_list_w;
      label_w = cat_label;
      class_names = (char **) catalog->class_names();
      nclasses = catalog->nclasses();
      list_label = XmStringCreateSimple((char *)catalog->catalog_file());
    }
    else {
      list_w = rec_list_w;
      label_w = rec_label;
      class_names = (char **) classifier->class_names();
      nclasses = classifier->nclasses();
      list_label = XmStringCreateSimple((char *)classifier->file_name());
    }

    clear_da();
    XmListDeleteAllItems(list_w);       // clear current list
    clear_selection(listType);

    for (register short i = 0; i < nclasses; ++i)
      XmListAddItemUnselected(list_w,
                              XmStringCreateSimple(class_names[i]), 0);

  } // if listType

  //    show file-name on the list label
  XtVaSetValues(label_w,
                XmNlabelString, list_label,
                NULL);

  XmStringFree(list_label);

} // show_list()



/*****************************************************************
* HANDLE : handle_cat_showMarkExamples
*
* callback to show examples of a mark
*****************************************************************/

 static void
handle_cat_showMarkExamples(Widget, XtPointer client_data, void*)
{
  MarkEditor *ui = (MarkEditor *)client_data;
  ui->set_edit_mode(CAT_SHOW_EXAMPLES);
  const char *cur_selection = ui->curCatalogSelection();
  if (cur_selection)
    ui->show_examples(cur_selection);
  else
    ui->prompt_get_className();
}


/*****************************************************************
* HANDLE : handle_cat_addMarkExample
*
* callback to add MarkExample(s)
*****************************************************************/

static void
handle_cat_addMarkExample(Widget, XtPointer client_data, void*)
{
  MarkEditor *ui = ((MarkEditor *)client_data);
  ui->set_edit_mode(CAT_ADD_EXAMPLES);
  const char *cur_selection = ui->curCatalogSelection();
  if (cur_selection)
    ui->catalog_addExamples(cur_selection);
  else
    ui->prompt_get_className();
}


/*****************************************************************
* HANDLE : handle_cat_setRepExample
*
* callback to add
*****************************************************************/

static void
handle_cat_setRepExample(Widget, XtPointer client_data, void*)
{
  MarkEditor *ui = ((MarkEditor *)client_data);
  ui->set_edit_mode(CAT_SET_REP_EXAMPLE);
  const char *cur_selection = ui->curCatalogSelection();
  if (cur_selection)
    ui->catalog_addExamples(cur_selection);
  else
    ui->prompt_get_className();
}

/*****************************************************************
* HANDLE : handle_cat_removeClass
*
* callback to remove a class from a catalog
*****************************************************************/

static void
handle_cat_removeClass(Widget, XtPointer client_data, void*)
{
  ((MarkEditor *)client_data)->catalog_removeClass();
}


/*****************************************************************/
/* prompt_get_className()                                        */
/*****************************************************************/

void
MarkEditor::prompt_get_className()
{
  if ( (edit_mode == CAT_SET_REP_EXAMPLE) ||
       (edit_mode == CAT_SHOW_EXAMPLES) ||
       (edit_mode == CAT_ADD_EXAMPLES) )
    if(!catalog) {
      print_message("No Catalog selected. Select one first.", 'n');
      return;
    }

  if ( (edit_mode == CLASSIFIER_SET_REP_EXAMPLE) ||
       (edit_mode == CLASSIFIER_DRAWADD_MARKS) )
    if (!classifier) {
      print_message("No Classifier selected. Select one first.", 'n');
      return;
    }
```

```
  XmString sel_str = XmStringCreateSimple("Enter Class Name:");

  //    create the prompt dialog box
  prompt_dialog = XmCreatePromptDialog(toplevel_w, "prompt_get_className",
                                       NULL, 0);
  XtVaSetValues(prompt_dialog,
                XmNselectionLabelString, sel_str,
                XmNautoUnmanage,         False,
                NULL);

  XmStringFree(sel_str);

  XtVaSetValues(XtParent(prompt_dialog),
                XmNtitle, "Get Class Name",
                NULL);

  XtAddCallback(prompt_dialog, XmNokCallback,
                (XtCallbackProc)prompt_get_className_ok_callback,
                (XtPointer)this);

  XtAddCallback(prompt_dialog, XmNcancelCallback,
                (XtCallbackProc)file_cancel_callback, (XtPointer)this);

  XtUnmanageChild(XmSelectionBoxGetChild(prompt_dialog,
                                         XmDIALOG_HELP_BUTTON));
  XtManageChild(prompt_dialog);
  XtPopup(XtParent(prompt_dialog), XtGrabNone);

} //prompt_get_className()



/*****************************************************************
* HANDLE : prompt_get_className_ok_callback()
* callback
*****************************************************************/

static void
prompt_get_className_ok_callback(Widget prompt_w, XtPointer client_data,
                                 XmSelectionBoxCallbackStruct *cbs)
{
  char  *class_name;

  if( !XmStringGetLtoR(cbs->value, charset, &class_name) ){
    printf("Internal error\n");
    XtDestroyWidget(prompt_w);          //prompt_w no longer needed
    return;
  }

  XtDestroyWidget(prompt_w);    //prompt_w no longer needed
  MarkEditor *ui = ((MarkEditor *)client_data);

  MarkEditorEditMode edit_mode = ui->get_edit_mode();

  if ((edit_mode == CAT_ADD_EXAMPLES) || (edit_mode == CAT_SET_REP_EXAMPLE))
    ui->catalog_addExamples(class_name);

  else if (edit_mode == CLASSIFIER_SET_REP_EXAMPLE)
    ui->classifier_setRepExample(class_name);

  else if (edit_mode == CLASSIFIER_DRAWADD_MARKS)
    ui->classifier_drawAddMarks(class_name);

  else if (edit_mode == CAT_SHOW_EXAMPLES)
    ui->show_examples(class_name);

} //prompt_get_className_ok_callback()


/*****************************************************************/
/* show_examples()                                               */
/*****************************************************************/

void
MarkEditor::show_examples(const char *class_name)
{
  edit_mode = NONE;

  if (!catalog) {
    print_message("No Catalog selected. Select one first.", 'n');
    return;
  }
  if (!class_name)
    return;

  if (!catalog->class_member(class_name))
    return;

  MarkExampleSet *exs = catalog->example_set(class_name);
  if (!exs) return;

  unsigned int nexamples = exs->nexamples();
  register MarkExample *ex;
  register unsigned int i;

  clear_da();

  for (i = 0; i < nexamples; ++i) {
    ex = exs->get_example(i);
    draw_example(ex, exs->class_name());
  }

  //    copy the widow contents into the pixmap
  XCopyArea(da_display, da_win, da_pixmap, da_gc, 0, 0, da_width, da_height, 0, 0);

} // show_examples()



/*****************************************************************/
/* catalog_addExamples()                                         */
/*****************************************************************/

void
MarkEditor::catalog_addExamples(const char *class_name)
{
  if (!catalog) {
    print_message("No Catalog selected. Select one first.", 'n');
    edit_mode = NONE;
    return;
  }
  if (!class_name) {
    edit_mode = NONE;
    return;
  }

  strcpy (example_className, class_name);

  int index = catalog->name_to_index(example_className);

  if (edit_mode == CAT_SET_REP_EXAMPLE) {
    if (index < 0)
```

```
    {
      edit_mode = NONE;
      print_message("Cann't set Rep. Example of a new class.", 'n');
      print_message("First add the new class by adding examples.", 'c');
      return;
    }
    else
      print_message ("Draw REPRESENTATIVE EXAMPLE in the drawing area.", 'n');

  }
  else if(edit_mode == CAT_ADD_EXAMPLES) {
    if (index >= 0)
    {
      cur_exampleSet = catalog->example_set(example_className);
      print_message ("Draw EXAMPLES in the drawing area.", 'n');
      print_message ("Press ANY KEY (in the drawing area) when done.", 'c');
    }
    else  // new class-name
    {
      Widget dialog = XmCreateQuestionDialog(toplevel_w,
                                             "question_add_newClass",
                                             NULL, 0);
      char msg[] = " is a new class. Add to the Catalog?";
      char *final_msg = new char [strlen(example_className) + strlen(msg) + 2];
      strcpy(final_msg, example_className);
      strcat(final_msg, msg);
      XmString msg_str = XmStringCreateSimple(final_msg);
      XmString ok_str = XmStringCreateSimple("Yes");

      XtVaSetValues(dialog,
                    XmNmessageString, msg_str,
                    XmNokLabelString, ok_str,
                    NULL);
      XtUnmanageChild(XmMessageBoxGetChild(dialog, XmDIALOG_HELP_BUTTON));
      XtAddCallback(dialog, XmNokCallback, cat_addNewClass_callback,
                    (XtPointer)this);
      XtAddCallback(dialog, XmNcancelCallback, file_cancel_callback,
                    (XtPointer)this);

      XmStringFree(msg_str);
      XmStringFree(ok_str);

      XtManageChild(dialog);
      XtPopup(XtParent(dialog), XtGrabNone);
    }

  } // else if

  clear_da();

} // catalog_addExamples()



/*****************************************************************
* HANDLE : cat_addNewClass_callback
*
* callback to add new class to a catalog
*****************************************************************/

static void
cat_addNewClass_callback(Widget, XtPointer client_data , void *cbs)
{
  MarkEditor *ui = (MarkEditor *) client_data;
  ui->catalog_addNewClass();
}
```

```
/*****************************************************************/
/* catalog_addNewClass()                                         */
/*****************************************************************/

void
MarkEditor::catalog_addNewClass()
{
  if (example_className) {

    //  add to list if new class-name:
    XmString str = XmStringCreateSimple(example_className);
    XmListAddItemUnselected(cat_list_w, str, 0);
    XmStringFree(str);
    cur_exampleSet = catalog->example_set(example_className);
    clear_da();
    print_message ("Draw EXAMPLES in the drawing area.", 'n');
  }
  else {
    printf("catalog_addNewClass: something wierd\n");
    edit_mode = NONE;
  }


} //catalog_addNewClass


/*****************************************************************/
/* catalog_removeClass()                                         */
/*****************************************************************/

void
MarkEditor::catalog_removeClass()
{
  if (!catalog) return;

  //    only removes the currently selected classname in the list

  const char *curClass = curCatalogSelection();
  if (!curClass) {
    print_message("Select a Class to Remove from the Catalog List.", 'n');
    return;
  }
  if (catalog->remove_class(curClass)) {
    XmString str = XmStringCreateSimple((char *)curClass);
    XmListDeleteItem(cat_list_w, str);

    XmListDeleteItem(cat_list_w, str);
    delete _cat_selection.item; _cat_selection.item = NULL;
    _cat_selection.selected_item_count--;

    XmStringFree(str);
    clear_da();
  }

  return;

} //catalog_removeClass



/*****************************************************************
* HANDLE : handle_rec_setRepExample
*
* callback to add
*****************************************************************/
```

```
static void
handle_rec_setRepExample(Widget, XtPointer client_data, void*)
{
  MarkEditor *ui = ((MarkEditor *)client_data);
  ui->set_edit_mode(CLASSIFIER_SET_REP_EXAMPLE);
  const char *cur_selection = ui->curClassifierSelection();
  if (cur_selection)
    ui->classifier_setRepExample(cur_selection);
  else
    ui->prompt_get_className();
}


/*****************************************************************/
/* classifier_setRepExample()                                    */
/*****************************************************************/

void
MarkEditor::classifier_setRepExample(const char *class_name)
{
  if (!classifier) {
    print_message("No Classifier selected. Select one first.", 'n');
    return;
  }
  if (!class_name)
    return;

  strcpy (example_className, class_name);

  //    do not set rep-example of a new class.
  if (classifier->new_class(class_name)) return;

  clear_da();
  print_message ("Draw REPRESENTATIVE EXAMPLE in the drawing area.", 'n');

} // classifier_setRepExample()


/*****************************************************************
* HANDLE : handle_rec_addMarks
*
* callback to add
*****************************************************************/

static void
handle_rec_addMarks(Widget, XtPointer client_data, void*)
{
  ((MarkEditor *)client_data)->rec_setup_addMarks();
}


/*****************************************************************/
/* rec_setup_addMarks()                                          */
/*****************************************************************/

void
MarkEditor::rec_setup_addMarks()
{
  if (!classifier) {
    print_message("No Classifier selected. Select one first.", 'n');
    return;
  }
  if (!catalog) {
    print_message("No Catalog to selected. Select one to chose marks from.", 'n');
    return;
  }

  print_message("Select marks from the catalog. Press <Return> when done.", 'n');

  edit_mode = CLASSIFIER_ADD_MARKS;

} // rec_setup_addMarks()



/*****************************************************************/
/* rec_addMarks()                                                */
/*****************************************************************/

void
MarkEditor::rec_addMarks()
{
  if (!classifier || !_cat_selection.selected_item_count) {
    print_message("NO classifier or NO selection from Catalog", 'n');
    return;
  }

  char **items = _cat_selection.selected_items;
  int nitems = _cat_selection.selected_item_count;
  classifier->add_examples((const char **)items, nitems, catalog);

  //    copy the rep-examples from catalog to the classifier
  for (int ns = 0; ns < nitems; ++ns)
  {
    //  add the mark's rep-example to classifier if it exists
    const MarkExample *rep = catalog->rep_example((char *)items[ns]);
    if(rep)
      classifier->set_rep_example(items[ns], *rep);
  }

  //    update the list widget

  XmString str;
  for (int i = 0; i < _cat_selection.selected_item_count; ++i) {
    str = XmStringCreateSimple((char *)items[i]);
    XmListAddItemUnselected(rec_list_w, str, 0);
    XmStringFree(str);
  }

} // rec_addMarks()



/*****************************************************************
* HANDLE : handle_rec_drawAddMarks
*
* callback to add marks to a class by drawing
*****************************************************************/

static void
handle_rec_drawAddMarks(Widget, XtPointer client_data, void*)
{
  MarkEditor *ui = ((MarkEditor *)client_data);
  const char* curSel = ui->curClassifierSelection();
  ui->set_edit_mode(CLASSIFIER_DRAWADD_MARKS);

  if(curSel)
    ui->classifier_drawAddMarks(curSel);
  else
    ui->prompt_get_className();

}
```

```
/*****************************************************************/
/* classifier_drawAddMarks()                                     */
/*****************************************************************/

void
MarkEditor::classifier_drawAddMarks(const char *class_name)
{
  if (!classifier || !class_name) {
    edit_mode = NONE;
    return;
  }

  strcpy (example_className, class_name);

  if(classifier->new_class(class_name))
  {
    Widget dialog = XmCreateQuestionDialog(toplevel_w,
                                           "question_add_newClass",
                                           NULL, 0);
    char msg[] = " is a new class. Add to the Classifier?";
    char *final_msg = new char [strlen(example_className) + strlen(msg) + 2];
    strcpy(final_msg, example_className);
    strcat(final_msg, msg);
    XmString msg_str = XmStringCreateSimple(final_msg);
    XmString ok_str = XmStringCreateSimple("Yes");

    XtVaSetValues(dialog,
                  XmNmessageString, msg_str,
                  XmNokLabelString, ok_str,
                  NULL);
    XtUnmanageChild(XmMessageBoxGetChild(dialog, XmDIALOG_HELP_BUTTON));
    XtAddCallback(dialog, XmNokCallback, rec_addNewClass_callback,
                  (XtPointer)this);
    XtAddCallback(dialog, XmNcancelCallback, file_cancel_callback,
                  (XtPointer)this);

    XmStringFree(msg_str);
    XmStringFree(ok_str);

    XtManageChild(dialog);
    XtPopup(XtParent(dialog), XtGrabNone);

  } // new class

  else {
    print_message ("Draw EXAMPLES in the drawing area.", 'n');
    print_message ("Press ANY KEY (in the drawing area) when done.", 'c');
    clear_da();
  }


} // classifier_drawAddMarks()



/*****************************************************************
* HANDLE : rec_addNewClass_callback
*
* callback to add new class to a classifier
*****************************************************************/

static void
 rec_addNewClass_callback(Widget, XtPointer client_data, void*)
{
  MarkEditor *ui = (MarkEditor *) client_data;
  ui->classifier_addNewClass();
}


/*****************************************************************/
/* classifier_addNewClass()                                      */
/*****************************************************************/

void
MarkEditor::classifier_addNewClass()
{
  if (example_className) {

    XmString str = XmStringCreateSimple(example_className);
    XmListAddItemUnselected(rec_list_w, str, 0);
    clear_da();
    print_message ("Draw EXAMPLES in the drawing area.", 'n');
    XmStringFree(str);
  }
  else {
    edit_mode = NONE;
  }

} // classifier_addNewClass()


/*****************************************************************
* HANDLE : handle_rec_removeClass
*
* callback to remove a class from a classifier
*****************************************************************/

static void
handle_rec_removeClass(Widget, XtPointer client_data, void*)
{
  ((MarkEditor *)client_data)->classifier_removeClass();

}


/*****************************************************************/
/* classifier_removeClass()                                      */
/*****************************************************************/

void
MarkEditor::classifier_removeClass()
{
  if (!classifier) return;

  //    only removes the currently selected classname in the list

  const char *curClass = curClassifierSelection();
  if (!curClass) {
    print_message("Select a Class to Remove from the Classifier List.", 'n');
    return;
  }
  if (classifier->remove_class(curClass)) {
    XmString str = XmStringCreateSimple((char *)curClass);

    XmListDeleteItem(rec_list_w, str);
    delete _rec_selection.item; _rec_selection.item = NULL;
    _rec_selection.selected_item_count--;

    XmStringFree(str);
    clear_da();
  }

  return;
```

```
} // classifier_removeClass()



/*****************************************************************
* HANDLE : handle_rec_train
*
* callback to add
*****************************************************************/

static void
handle_rec_train(Widget, XtPointer client_data, void*)
{
  ((MarkEditor *)client_data)->rec_train();
}


/*****************************************************************/
/* rec_train()                                                   */
/*****************************************************************/

void
MarkEditor::rec_train()
{
  if (!classifier) {
    print_message("No Classifier to train. Select one first.", 'n');
    return;
  }
  if (classifier->nclasses())
    classifier->train();

} // rec_train()


/*****************************************************************
* HANDLE : handle_rec_classifyExamples
*
* callback to test (classify user\'s examples)
*****************************************************************/

static void
handle_rec_classifyExamples(Widget, XtPointer client_data, void*)
{
  MarkEditor * ui = (MarkEditor *)client_data;
  ui->classifier_classifyExamples();

}

/*****************************************************************/
/* classifier_classifyExamples()                                 */
/*****************************************************************/

void
MarkEditor::classifier_classifyExamples()
{
  if (!classifier) {
    set_edit_mode(NONE);
    print_message("No Classifier currently selected.", 'n');
    return;
  }
  if (!classifier->trained()) {
    set_edit_mode(NONE);
    print_message("Classifier NOT TRAINED. Train it first.", 'n');
    return;
  }
  if (!classifier->nclasses()) {
    set_edit_mode(NONE);
    print_message("Classifier Empty. Nothing to Classify.", 'n');
    return;
  }

  set_edit_mode(CLASSIFIER_CLASSIFY_EXAMPLES);
  print_message("Draw Examples to Classify in the Drawing Area.", 'n');
  print_message ("Press ANY KEY (in the drawing area) when done.", 'c');
  clear_da();

} // classifier_classifyExamples()


/*****************************************************************
* HANDLE : handle_help
*
* callback to provide help
*****************************************************************/

static void
handle_help(Widget, XtPointer client_data , void*)
{
  ((MarkEditor *)client_data)->help_tutorial();
}

/*****************************************************************/
/* help_tutorial()                                               */
/*****************************************************************/

void
MarkEditor::help_tutorial()
{
  Widget        tutorial_dialog, scroll_w, pushb_w;
  XmString      x_help_text;
  FILE*         tutorial_fp;
  short         len = 0;        //line length
  long          file_length;

  if (!(tutorial_fp = fopen("Mark.help", "r")) )
    return;

  fseek(tutorial_fp, 0L, 2); //position byte_ptr to EOF
  file_length = ftell(tutorial_fp);  //get cur byte's offset
  fseek(tutorial_fp, 0L, 0); //point back to start of file

  //    create a buffer of of the file_length size
  char* buf = new char[file_length + 2];
  strcpy(buf, "");                  // make the first character empty
  char* start_buf = buf;            // save the starting addrs of buffer

  char* line = new char[MAXLINE];  // temp string to store a line

  //    repeat until all lines of the file have been read
  while( fgets(line, MAXLINE, tutorial_fp) ){
    strcat(buf, line);
    len = strlen(line);     // get the length of the new line fetched
    buf += len;             // update the buf ptr
  }

  buf = start_buf;

  x_help_text = XmStringCreateLtoR(start_buf, XmSTRING_DEFAULT_CHARSET);


  shell_w = XtVaCreatePopupShell("shell_help", xmDialogShellWidgetClass,
                                 get_top_shell(toplevel_w),
```

```
                                XmNdeleteResponse, XmDESTROY,
                                NULL);

   pane_w = XtVaCreateWidget("pane_help", xmPanedWindowWidgetClass, shell_w,
                             XmNsashWidth, 1,
                             XmNsashHeight, 1,
                             NULL);


   scroll_w = XtVaCreateManagedWidget("help_scrolled_win",
                                      xmScrolledWindowWidgetClass, pane_w,
                                      XmNscrollingPolicy, XmAUTOMATIC,
                                      NULL);

   Widget label_w = XtVaCreateManagedWidget("help_label",
                                     xmLabelWidgetClass, scroll_w,
                                     XmNlabelString, x_help_text,
                                     NULL);

   pushb_w = XtVaCreateManagedWidget("cancel",
                                     xmPushButtonWidgetClass, pane_w,
                                     NULL);

   XtAddCallback(pushb_w, XmNactivateCallback, destroy_shell, shell_w);

   XmStringFree(x_help_text);

   delete buf;
   delete line;

   XtManageChild(pane_w);
   XtPopup(shell_w, XtGrabNone);

} // help_tutorial

/*****************************************************************
* HANDLE : handle_da_callback
*
* callback for inputs to drawing area
*****************************************************************/

static void
handle_da_callback(Widget, XtPointer client_data, void *cbs)
{
   ((MarkEditor*)client_data)->da_activity((XmDrawingAreaCallbackStruct *)cbs);

}


/*****************************************************************/
/* da_activity()                                                 */
/* callback for expose/inputs to drawing area                    */
/*****************************************************************/

void
MarkEditor::da_activity(XmDrawingAreaCallbackStruct *da_cbs)
{
   XEvent  *da_event = da_cbs->event;
   static int npoints;
   static MarkPoint curp, lastp;

   if(da_cbs->reason == XmCR_INPUT)
   {
     if ( (edit_mode == CAT_ADD_EXAMPLES) ||
          (edit_mode == CAT_SET_REP_EXAMPLE) ||
          (edit_mode == CLASSIFIER_SET_REP_EXAMPLE) ||
          (edit_mode == CLASSIFIER_DRAWADD_MARKS) ||
          (edit_mode == CLASSIFIER_CLASSIFY_EXAMPLES) )
     {
       if(da_event->xany.type == ButtonPress){
         cur_example = new MarkExample;
         curp.x = da_event->xbutton.x;
         curp.y = da_event->xbutton.y;
         // curp.time = da_event->xbutton.time;
         curp.time = 0;
         lastp = curp;
         cur_example->add_point(curp);
       }

       else if(da_event->xany.type == MotionNotify){
         lastp = curp;
         curp.x = da_event->xbutton.x;
         curp.y = da_event->xbutton.y;
         // curp.time = da_event->xbutton.time;
         curp.time = 0;
         cur_example->add_point(curp);

         // draw the mark, join last to the current point
         XSetFunction(da_display, da_gc, GXcopy);
         XDrawLine(da_display, da_win, da_gc, lastp.x, lastp.y,
                   curp.x, curp.y);
       }

       else if(da_event->xany.type == ButtonRelease){
         lastp = curp;
         curp.x = da_event->xbutton.x;
         curp.y = da_event->xbutton.y;
         // curp.time = da_event->xbutton.time;
         curp.time = 0;
         cur_example->add_point(curp);

         // draw the mark, join last to the current point
         XDrawLine(da_display, da_win, da_gc, lastp.x, lastp.y,
                   curp.x, curp.y);
         XFlush(da_display);
         XCopyArea(da_display, da_win, da_pixmap, da_gc,
                   0, 0, da_width, da_height, 0, 0);

         if (edit_mode == CAT_ADD_EXAMPLES) {
           cur_example->set_example_name (example_className);
           cur_exampleSet->add_example(*cur_example);
           catalog->set_change_status(1);
         }
         else if (edit_mode == CAT_SET_REP_EXAMPLE) {
           catalog->set_rep_example(example_className, *cur_example);
           edit_mode = NONE;      // allow to draw only once
         }
         else if (edit_mode == CLASSIFIER_SET_REP_EXAMPLE) {
           classifier->set_rep_example(example_className, *cur_example);
           edit_mode = NONE;
         }
         else if (edit_mode == CLASSIFIER_DRAWADD_MARKS) {
           classifier->add_example(example_className, *cur_example);
           classifier->train();
         }
         else if (edit_mode == CLASSIFIER_CLASSIFY_EXAMPLES) {
           const char *className = classifier->classify_example(cur_example);
           if (className) {
             print_message((char *)className, 'n');
           }
           else
             print_message("Example Not Recognized.", 'n');
         }
```

```
        delete cur_example;

      } // else if buttonRelease

    } // if edit mode

    if ((da_event->xany.type == KeyRelease)) {
      // printf("key code = %d \n", da_event->xkey.keycode);
      if (da_event->xkey.keycode == KEYCODE_D) {
        edit_mode = NONE;
        clear_da();
      }
    }

  } // reason = input

  else if(da_cbs->reason == XmCR_EXPOSE) {
    da_win = XtWindow(da_w);
    XClearWindow(da_display, da_event->xany.window);

    //  copy the pixmap contents into the window
    XCopyArea(da_display, da_pixmap, da_win, da_gc, 0, 0,
              da_width, da_height, 0, 0);
  }

  else if (da_cbs->reason == XmCR_RESIZE) {

    Pixmap old_pixmap = da_pixmap;
    unsigned short old_width = da_width;
    unsigned short old_height = da_height;

    da_win = XtWindow(da_w);

    if (!da_win) return;

    XtVaGetValues(da_w, XmNwidth, &da_width, XmNheight, &da_height, NULL);

    //  paint the window clear
    XSetForeground(da_display, da_gc, da_bg );

    //  clear the pixmap with bg color
    XFillRectangle(da_display, da_win, da_gc, 0, 0, da_width, da_height);

    //  revert fg to normal
    XSetForeground(da_display, da_gc, da_fg );

    //  copy the old pixmap contents into the window
    XCopyArea(da_display, old_pixmap, da_win, da_gc,
              0, 0, old_width, old_height, 0, 0);

    // create a new pixmap of the new size
    da_pixmap = XCreatePixmap(da_display, RootWindowOfScreen(da_screen),
                              da_width, da_height,
                              DefaultDepthOfScreen(da_screen) );

    XCopyArea(da_display, old_pixmap, da_pixmap, da_gc,
              0, 0, old_width, old_height, 0, 0);

    XFreePixmap(da_display, old_pixmap);
  }

} //da_activity()


/*****************************************************************/
/* draw_example()                                                */
/*****************************************************************/

void
MarkEditor::draw_example(const MarkExample *ex, const char *name)
{
  if (!ex || ex->empty()) return;

  unsigned int npoints;
  register int i;
  register MarkPoint p1, p2;

//  XtVaGetValues(da_w,
//                XmNwidth, &da_width,
//                XmNheight, &da_height,
//                NULL);
//  printf("da_width = %d, da_height = %d\n", da_width, da_height);


  MarkBBox bbox;
  ex->bbox(bbox);

  double max_x = bbox.max_x;
  double max_y = bbox.max_y;

  double mag_x = da_width/(max_x + 50);
  double mag_y = da_height/(max_y + 50);

  MarkPoint *points = ex->points(&npoints);

  for (i = 0; i < (npoints - 1); ++i)
  {
    p1 = points[i];
    p2 = points[i + 1];
    int x1 = mag_x * p1.x;
    int x2 = mag_x * p2.x;
    int y1 = mag_y * p1.y;
    int y2 = mag_y * p2.y;
    XDrawLine(da_display, da_win, da_gc, x1, y1, x2, y2);
  }

  //    print the name
  int str_len = strlen(name);
  if (!name || !str_len) return;
  int width = XTextWidth(font_struct, name, str_len);
  XDrawString(da_display, da_win, da_gc, NAME_START_X, NAME_START_Y,
              name, str_len);

} // draw_example()



/*****************************************************************/
/* print_message()                                               */
/*****************************************************************/

void
MarkEditor::print_message (char *message, char new_or_cont)
{
  static XmTextPosition text_position;
  static int message_id;
  char final_str[1000];

  if (message)
  {
    if(new_or_cont == 'n')
      sprintf(final_str, "\n[%d] ", ++message_id);
    else
      sprintf(final_str, "    \n");
```

```
    strcat(final_str, message);
    XmTextInsert (message_w, text_position, final_str);
    text_position += strlen(final_str);
    XtVaSetValues (message_w, XmNcursorPosition, text_position, NULL);
    XmTextShowPosition (message_w, text_position);
  }

} // print_message()



/*****************************************************************
* HANDLE : handle_list
*
* callback to list
*****************************************************************/

void
handle_list (Widget w, XtPointer client_data, void *cb)
{
  XmListCallbackStruct *cbs = (XmListCallbackStruct *)cb;
  MarkEditor *ui = ((MarkEditor *)client_data);

  if (cbs->reason == XmCR_DEFAULT_ACTION) {
    // gets here when user presses return after selecting a list
    // of marks from the catalog to add to a classifier.

    MarkEditorEditMode mode = ui->get_edit_mode();
    if (mode != CLASSIFIER_ADD_MARKS) return;

    ui->set_edit_mode(NONE);
    ui->update_selection(w, cbs);
    ui->rec_addMarks();
  }

  else {
    // ui->set_edit_mode(NONE);
    ui->update_selection(w, cbs);
  }

}// handle_list()



/*****************************************************************/
/* update_selection()                                            */
/*****************************************************************/

void
MarkEditor::update_selection(Widget w, XmListCallbackStruct *cbs)
{
  char *choice;
  ListSelection *_selection;

  if (w == cat_list_w) {
    _selection = &_cat_selection;
    clear_selection('c');              // clean up prev. selection
  }
  else if (w == rec_list_w) {
    _selection = &_rec_selection;
    clear_selection('r');              // clean up prev. selection
  }
  else {
    printf("update_selection: strange widget\n");
    return;
  }

  //    ALLOCATE SPACE FOR NEW SELECTION

  if (cbs->selected_item_count) {
    XmStringGetLtoR(cbs->item, charset, &choice);
    _selection->item = new char[strlen(choice) + 1];
    strcpy (_selection->item, choice);
    XtFree(choice);
    _selection->item_position = cbs->item_position;
  }

  _selection->selected_item_positions = new int[cbs->selected_item_count];

  //    ASSIGN SELECTION

  for (int i = 0; i < cbs->selected_item_count; ++i) {
    XmStringGetLtoR(cbs->selected_items[i], charset, &choice);
    _selection->selected_items[i] = new char [strlen(choice) + 1];
    strcpy(_selection->selected_items[i], choice);
    XtFree(choice);

    _selection->selected_item_positions[i] = cbs->selected_item_positions[i];
  }
  _selection->selected_item_count = cbs->selected_item_count;

  MarkExample *ex = NULL;

  if (w == cat_list_w){
    if (catalog)
      ex = (MarkExample *) catalog->rep_example(_selection->item);
  }
  else if (w == rec_list_w) {
    if (classifier)
      ex = (MarkExample *) classifier->rep_example(_selection->item);
  }

  clear_da();
  draw_example(ex, _selection->item);
  XCopyArea(da_display, da_win, da_pixmap, da_gc,
            0, 0, da_width, da_height, 0, 0);

} //update_selection()


/*****************************************************************/
/* get_top_shell()                                               */
/*****************************************************************/

Widget
get_top_shell(Widget w)
{
  while (w && !XtIsWMShell(w))
    w = XtParent(w);
  return w;
}

/*****************************************************************/
/* destroy_shell()                                               */
/*****************************************************************/

void
destroy_shell(Widget w, void *shell, void *)
{
  XtDestroyWidget((Widget)shell);
}
```

```
/*****************************************************************
* HANDLE : file_cancel_callback()
*****************************************************************/

static void
file_cancel_callback(Widget w, XtPointer client_data, void *)
{
  MarkEditor *ui = (MarkEditor*) client_data;
  if (ui)
  {
    MarkEditorEditMode edit_mode = ui->get_edit_mode();
    if ( (edit_mode == CAT_ADD_EXAMPLES) ||
         (edit_mode == CAT_SET_REP_EXAMPLE) ||
         (edit_mode == CLASSIFIER_SET_REP_EXAMPLE) ||
         (edit_mode == CLASSIFIER_DRAWADD_MARKS) ||
         (edit_mode == CAT_SHOW_EXAMPLES) )

      ui->set_edit_mode(NONE);
  }

  XtDestroyWidget(w);
  return;
}


/*****************************************************************
* catalog_index()
*****************************************************************/

int
MarkEditor::catalog_index(char *name)
{
  if (!name) return -1;

  for (int i = 0; i < UI_MAX_CATALOGS; ++i)
    if (_catalogs[i].catalog_file())
      if(!strcmp(name, _catalogs[i].catalog_file()) ) return i;

  return -1;

} // catalog_index()


/*****************************************************************
* classifier_index()
*****************************************************************/

int
MarkEditor::classifier_index(char *name)
{
  if (!name) return -1;

  for (int i = 0; i < UI_MAX_CLASSIFIERS; ++i)
    if(_classifiers[i].file_name())
      if(!strcmp(name, _classifiers[i].file_name()) ) return i;

  return -1;

} // classifier_index()
```

```
/*****************************************************************
* clear_da()
*****************************************************************/

void
MarkEditor::clear_da()
{
  //    clears drawing area
  da_win = XtWindow(da_w);
  XClearWindow (da_display, da_win);
  XCopyArea(da_display, da_win, da_pixmap, da_gc, 0, 0,
            da_width, da_height, 0, 0);
}

/*****************************************************************
* clear_selection()
*****************************************************************/

void
MarkEditor::clear_selection(char doctype)
{
  //    CLEAN UP PREVIOUS SELECTION

  ListSelection *_selection;

  if (doctype == 'c')
    _selection = &_cat_selection;
  else if (doctype == 'r')
    _selection = &_rec_selection;

  delete _selection->item; _selection->item = NULL;

  for (register short i = 0; i < _selection->selected_item_count; ++i) {
    delete _selection->selected_items[i];
    _selection->selected_items[i] = NULL;
  }

  delete _selection->selected_item_positions;
  _selection->selected_item_positions = NULL;


} // clear_selection()
```

```
/*****************************************************************************/
/* MarkEditorMain.C                                                          */
/*                                                                           */
/* Author:  Lalit K. Agarwal, Brown University                               */
/* Date  :  May 1994                                                         */
/*****************************************************************************/


#include "MarkEditor.H"
#include "MarkDefaults.H"

char _zdebug_flag[128];

main (int argc, char **argv)
{
  XtAppContext app_context;
  Widget toplevel;

  toplevel = XtVaAppInitialize(
                               &app_context,    /* Application context */
                               "Mark",          /* App class, name of resource file */
                               NULL, 0,         /* command line option list */
                               &argc, argv,     /* command line args */
                               fallback_res,    /* for missing app-defaults file */
                               NULL);           /* terminate varargs list */

  XtVaSetValues(toplevel, XmNunitType, XmPIXELS, NULL);

  //    constructor creates the Editor
  MarkEditor ui = MarkEditor (toplevel);

  XtRealizeWidget(toplevel);
  XtAppMainLoop(app_context);

  return 1;

} // main()
```

```
/*****************************************************************************/
/* MarkExample.C                                                             */
/*                                                                           */
/* Methods for class MarkExample                                             */
/*                                                                           */
/* Author:  Lalit K. Agarwal, Brown University                               */
/* Date  :  May, 1994                                                        */
/*****************************************************************************/

#include <assert.h>
#include <string.h>
#include <iostream.h>
#include <stdio.h>
#include <stdlib.h>

#include "MarkExample.H"

#define EXAMPLE_SIZE_INCR 50

/*****************************************************************************/
/* MarkExample: Default Constructor                                          */
/*****************************************************************************/

MarkExample::MarkExample(uint npoints, MarkPoint *initArray,
                         const char* initName)
{
  register short i;

  init_array (npoints);

  _npoints = 0;  // if null input-array, then no actual points

  if(initArray) {       // if given a valid initialization array of points
    _npoints = npoints;
    for (i = 0; i < _npoints; i++)
      _pointArray[i] = initArray[i];    // copy points
  }

  init_name (initName);

} //constructor



/*****************************************************************************/
/* MarkExample: Member-wise-initialization constructor                       */
/*****************************************************************************/

MarkExample::MarkExample(const MarkExample &rhs)
{
  init_array (rhs.size());

  _npoints = rhs.npoints();

  register uint i;
  for (i = 0; i < _npoints; ++i)
    _pointArray[i] = *rhs.get_point(i);         // copy points

  init_name (rhs.name());

} //MarkExample(const MarkExample &)



/*****************************************************************************/
/* MarkExample:destructor                                                    */
/*****************************************************************************/

MarkExample::~MarkExample()
{
  delete [] _pointArray; _pointArray = NULL;
  delete _name; _name = NULL;

} //destructor


/*****************************************************************************/
/* operator =                                                                */
/*****************************************************************************/

MarkExample&
MarkExample::operator = (const MarkExample &rhs)
{
  if (this == &rhs)
    return *this;

  MarkPoint *oldArray = _pointArray;

  _size = rhs.size();
  _pointArray = new MarkPoint[_size];
  assert(_pointArray);

  _npoints = rhs.npoints();
  for (register short i = 0; i < _npoints; ++i)
    _pointArray[i] = *rhs.get_point(i);       // copy points

  delete oldArray;

  _name = NULL;
  const char *rhsName = rhs.name();
  if (rhsName) {
    _name = new char [strlen(rhsName) + 1];  // 1 for null char at end
    assert (_name);
    strcpy (_name, rhsName);
  }

  return *this;

} //operator = ()



/*****************************************************************************/
/* over_write                                                                */
/*****************************************************************************/

void
MarkExample::over_write(uint npoints,
                        MarkPoint *newArray, char *newName)
{
  register short i;
  MarkPoint *oldArray = _pointArray;

  _size = npoints;
  _pointArray = new MarkPoint[_size];
  assert(_pointArray);

  _npoints = 0;  // if null input-array, then no actual points

  if(newArray) {        // if given a valid new array of points
    _npoints = npoints;
    for (i = 0; i < _npoints; i++)
      _pointArray[i] = newArray[i];     // copy points from new array
```

```
  delete oldArray;

  set_example_name(newName);

} //over_write()



/*****************************************************************************/
/* add_point                                                                 */
/*****************************************************************************/

int
MarkExample::add_point(const MarkPoint& newPoint)
{
  if (_npoints == _size) {      // need to grow the point-array size
    MarkPoint *oldArray = _pointArray;
    uint       oldSize = _size;

    init_array (_size + EXAMPLE_SIZE_INCR);

    // copy elements of old array into new
    register short ix;
    for (ix = 0; ix < oldSize; ++ix)
      _pointArray[ix] = oldArray[ix];

    delete oldArray;
  }

  // copy the new point
  _pointArray[_npoints++] = newPoint;

  return _npoints;

} //add_point()



/*****************************************************************************/
/* set_example_name                                                          */
/*****************************************************************************/

void
MarkExample::set_example_name(const char *newName)
{
  delete _name;         // free memory occupied by the old name
  init_name (newName);

} //set_example_name()


/*****************************************************************************/
/* get_point: 1                                                              */
/*****************************************************************************/

int
MarkExample:: get_point(MarkPoint& inPoint, uint index) const
{
  if(index >= _npoints) return -1;
  inPoint = _pointArray[index];
  return index;

} //get_point(1)


/*****************************************************************************/
```

```
/* get_point: 2                                                              */
/*****************************************************************************/

MarkPoint*
MarkExample::get_point (uint index) const
{
  if(index >= _npoints) return NULL;
  return &_pointArray[index];

} //get_point(2)


/*****************************************************************************/
/* points                                                                    */
/*****************************************************************************/

MarkPoint*
MarkExample::points (uint *npoints) const
{
  if (!_npoints) {
    *npoints = 0;
    return NULL;
  }

  MarkPoint *reply = new MarkPoint [_npoints];
  assert (reply);

  // copy points into the reply
  register short ix;
  for (ix = 0; ix < _npoints; ++ix)
    reply[ix] = _pointArray[ix];

  *npoints = _npoints;

  return reply;

} //points()



/*****************************************************************************/
/* init_name                                                                 */
/*****************************************************************************/

void
MarkExample::init_name (const char *initName)
{
  _name = NULL;
  if (initName) {
    _name = new char [strlen(initName) + 1];  // extra 1 for null char at end
    assert (_name);
    strcpy (_name, initName);
  }

} //init_name()



/*****************************************************************************/
/* init_array                                                                */
/*****************************************************************************/

void
MarkExample::init_array (uint size)
{
  _size = size;
  _pointArray = new MarkPoint[_size];
```

```
  assert(_pointArray);

} //init_array()


/*****************************************************************************/
/* cleanup                                                                   */
/*****************************************************************************/

void
MarkExample::cleanup()
{
  delete _name; _name = NULL;

  delete [] _pointArray;
  init_array (EXAMPLE_DEFAULT_NUMPOINTS);
  _npoints = 0;

}


/*****************************************************************************/
/* bbox()                                                                    */
/*****************************************************************************/

int
MarkExample::bbox(MarkBBox &inbox) const
{
  if (!_npoints) {
    inbox.min_x = inbox.min_y = 0;
    inbox.max_x = inbox.max_y = 0;
    return 0;
  }

  register MarkPoint *p = &_pointArray[0];
  int max_x = 0; int max_y = 0;
  int min_x = p->x;
  int min_y = p->y;

  for (register int i = 0; i < _npoints; ++i)
  {
    p = &_pointArray[i];

    //  MIN_X
    if (min_x > p->x)
      min_x = p->x;

    //  MIN_Y
    if (min_y > p->y)
      min_y = p->y;

    //  MAX_X
    if (max_x < p->x)
      max_x = p->x;

    //  MAX_Y
    if (max_y < p->y)
      max_y = p->y;
  }

  inbox.min_x = min_x;
  inbox.min_y = min_y;
  inbox.max_x = max_x;
  inbox.max_y = max_y;

  return 1;
} // bbox()
```

```
/*****************************************************************************/
/* MarkExampleSet.C                                                          */
/*                                                                           */
/*                                                                           */
/*                                                                           */
/* Author:  Lalit K. Agarwal, Brown University                               */
/* Date   :  May 1994                                                        */
/*****************************************************************************/

#include "MarkExampleSet.H"
#include <assert.h>
#include <string.h>
#include <iostream.h>
#include <stdio.h>
#include <stdlib.h>

#define EXAMPLESET_SIZE_INCR 5

/*****************************************************************************/
/* MarkExampleSet: constructor                                               */
/*****************************************************************************/

MarkExampleSet::MarkExampleSet (uint nexamples, const MarkExample *initArray,
                                int index, const char *initName)
{
  register short i;

  init_array (nexamples);

  _nexamples = 0;  // if null initArray, then no actual examples

  //   Initialize only if a valid array of examples to initialize with

  if (initArray) {
    _nexamples = nexamples;
    for (i = 0; i < _nexamples; i++)
      _exampleArray[i] = initArray[i];  // overloaded operator "=" called
  }

  init_name (initName);
  _classIndex = index;
  _repExample = NULL;     // stays null until explicitly initialized

} //constructor



/*****************************************************************************/
/* MarkExampleSet: memberwise initialization constructor                     */
/* called during initialization by assignment, pass by value, return by-value*/
/*****************************************************************************/

MarkExampleSet::MarkExampleSet (const MarkExampleSet &rhs)
{
  init_array (rhs.size());
  _nexamples = rhs.nexamples();

  for (register short i = 0; i < _nexamples; i++)
    _exampleArray[i] = *rhs.get_example(i);

  init_name(rhs.class_name());

  _classIndex = rhs.index();

  if (rhs.rep_example()) {
    _repExample = new MarkExample;          // default constructor used
    *_repExample = *rhs.rep_example();  // assignment operator "="
  }

} //end memberwise-init constructor


/*****************************************************************************/
/* destructor                                                                */
/*****************************************************************************/

MarkExampleSet::~MarkExampleSet()
{
  delete [] _exampleArray; _exampleArray = NULL;
  delete _repExample; _repExample = NULL;
  delete _className;  _className = NULL;

} //destructor



/*****************************************************************************/
/* operator "=": explicit assignment after initialization                    */
/*****************************************************************************/

MarkExampleSet &
MarkExampleSet::operator = (const MarkExampleSet &rhs)
{
  if (this == &rhs)
    return *this;

  MarkExample   *oldArray = _exampleArray;

  init_array (rhs.size());

  for (register short i = 0; i < rhs.nexamples(); i++)
    _exampleArray[i] = *rhs.get_example(i);

  delete [] oldArray;

  _classIndex = rhs.index();
  init_name (rhs.class_name());

  delete _repExample; _repExample = NULL;

  if (rhs.rep_example()) {
    _repExample = new MarkExample;          // default constructor used
    *_repExample = *rhs.rep_example();  // assignment operator "="
  }

  _nexamples = rhs.nexamples();
  return *this;

} //operator "="


/*****************************************************************************/
/* set_class_name                                                            */
/*****************************************************************************/

void
MarkExampleSet::set_class_name(const char *newName)
{
  if (!newName) return;

  delete _className;            // need to free it anyway
  _className = new char [strlen(newName) + 1];
  assert (_className);
  _className = strcpy (_className, newName);
```

```
} //set_class_name()


/*****************************************************************************/
/* get_all_examples                                                          */
/*****************************************************************************/

MarkExample*
MarkExampleSet::get_all_examples(uint *nexamples) const
{
  MarkExample *reply = new MarkExample [_nexamples];
  assert (reply);

  for (register short i = 0; i < _nexamples; i ++)
    reply [i] = _exampleArray[i];       // copy examples

  *nexamples = _nexamples;

  return reply;

} //get_all_examples()


/*****************************************************************************/
/* get_example: 1                                                            */
/*****************************************************************************/

int
MarkExampleSet::get_example(MarkExample &inExample, uint index) const
{
  if (!_exampleArray || index >= _nexamples) return -1;
  inExample = _exampleArray[index];
  return index;

} //get_example(1)



/*****************************************************************************/
/* get_example: 2                                                            */
/*****************************************************************************/

MarkExample*
MarkExampleSet::get_example(uint index) const
{
  if (!_exampleArray || index >= _nexamples) return NULL;
  return &_exampleArray[index];

} //get_example(2)


/*****************************************************************************/
/* add_example                                                               */
/*****************************************************************************/

uint
MarkExampleSet::add_example(const MarkExample& newExample)
{
  // if need to grow the size of the array

  if (_nexamples == _size) {
    MarkExample *oldArray = _exampleArray;
    uint         oldSize = _size;

    init_array (_size + EXAMPLESET_SIZE_INCR);  // create a new array

    for (register short i = 0; i < oldSize; ++i) {
      _exampleArray[i] = oldArray[i];
    }

    delete [] oldArray;
  }

  // append the newExample at the end
  _exampleArray[_nexamples++] = newExample;

  return _nexamples;


} //add_example()


/*****************************************************************************/
/* set_rep_example                                                           */
/*****************************************************************************/

void
MarkExampleSet::set_rep_example (const MarkExample &newExample)
{
  delete _repExample;
  _repExample = new MarkExample;
  *_repExample = newExample;

} //set_rep_example()


/*****************************************************************************/
/* initName                                                                  */
/*****************************************************************************/

void
MarkExampleSet::init_name (const char *initName)
{
  _className = NULL;
  if (initName) {
    _className = new char [strlen(initName) + 1]; // extra 1 for null at end
    assert (_className);
    strcpy (_className, initName);
  }

} //init_name()


/*****************************************************************************/
/* cleanup                                                                   */
/*****************************************************************************/

void
MarkExampleSet::cleanup()
{
  _classIndex = -1;
  delete _repExample; _repExample = NULL;
  delete _className;  _className = NULL;

  _nexamples = 0;

  delete [] _exampleArray; _exampleArray = NULL;

  init_array(EXAMPLESET_DEFAULT_NUMEXAMPLES);

} // cleanup()
```

```
/*****************************************************************************/
/* MarkInterface.C                                                           */
/*                                                                           */
/* Methods of class MarkInterface                                            */
/*                                                                           */
/* Author:  Lalit K. Agarwal, Brown University                               */
/* Date   :  May 1994                                                        */
/*****************************************************************************/

#include "MarkExample.H"
#include "MarkInterface.H"
#include "MarkClassifier.H"

#include <Xm/DrawingA.h>
#include <Xm/DialogS.h>
#include <Xm/PanedW.h>
#include <Xm/Label.h>
#include <Xm/PushB.h>
#include <Xm/DrawnB.h>
#include <Xm/RowColumn.h>
#include <Xm/ScrolledW.h>

#include <stdio.h>
#include <stdlib.h>
#include <iostream.h>

#define EXAMPLE_WIDTH     100
#define EXAMPLE_HEIGHT    100
#define EXAMPLES_PER_COL  4
#define EXAMPLES_PER_ROW  4
#define SCROLLW_OFFSET    12
#define ENABLE_FG         "NavajoWhite"
#define ENABLE_BG         "DarkSlateGray"
#define NAME_START_X      10
#define NAME_START_Y      15

XmStringCharSet charset = XmSTRING_DEFAULT_CHARSET;

char _zdebug_flag[128]; // just ignore it.

/*****************************************************************************/
/*              Forward  Function Declarations                               */
/*****************************************************************************/

extern Widget
get_top_shell(Widget w);

static void
dismiss_shell_callback(Widget w, XtPointer shell, void *cbs);

static void
drawb_callback(Widget drawb_w, XtPointer client_data, void *cb);


/*********************************************/
/*      BtnMotionAction()                    */
/*********************************************/

static
void BtnMotionAction(Widget w, XEvent *event,
                     String *params,
                     Cardinal *num_params)
{
    MarkInterface *mi;
    XtVaGetValues(w, XmNuserData, &mi, NULL);
    if (mi) mi->eventHandler(w, event);

}/* end BtnMotionAction() */

/*********************************************/
/*      BtnUpAction()                        */
/*********************************************/

static
void BtnUpAction( Widget w, XEvent *event,
                  String *params,
                  Cardinal *num_params)
{
    MarkInterface *mi;
    XtVaGetValues(w, XmNuserData, &mi, NULL);
    if (mi) mi->eventHandler(w, event);

}/* end BtnUpAction() */

/*********************************************/
/*      BtnDownAction()                      */
/*********************************************/

static
void BtnDownAction( Widget w, XEvent *event,
                    String *params,
                    Cardinal *num_params)
{
    MarkInterface *mi;
    XtVaGetValues(w, XmNuserData, &mi, NULL);
    if (mi) mi->eventHandler(w, event);

}/* BtnDownAction() */


/*********************************************/
/*      HelpAction()                         */
/*********************************************/

static
void HelpAction( Widget w, XEvent *event,
                 String *params,
                 Cardinal *num_params)
{
    MarkInterface *mi;
    XtVaGetValues(w, XmNuserData, &mi, NULL);

    if (mi) mi->displayMarksRecognized();

}/* HelpAction() */


/*********************************************/
/*      ActivateAction()                     */
/*********************************************/

static
void ActivateAction( Widget w, XEvent *event,
                 String *params,
                 Cardinal *num_params)
{
    MarkInterface *mi;
    XtVaGetValues(w, XmNuserData, &mi, NULL);
    if (mi) mi->activateMarkingInteraction();

}/* ActivateAction() */


/*********************************************/
```

```
/*      DeactivateAction()                      */
/*********************************************/

static
void DeactivateAction( Widget w, XEvent *event,
                String *params,
                Cardinal *num_params)
{
    MarkInterface *mi;
    XtVaGetValues(w, XmNuserData, &mi, NULL);
    if (mi) mi->deactivateMarkingInteraction();

}/* DeactivateAction() */


/*********************************************/
/*      MarkAppInitialize()                  */
/*********************************************/

void
MarkAppInitialize(XtAppContext app_context)
{
  static XtActionsRec interface_actions[] =
  {
    { "btnDownAction",     BtnDownAction },
    { "btnUpAction",       BtnUpAction },
    { "btnMotionAction",   BtnMotionAction },
    { "helpAction",        HelpAction },
    { "activateAction",    ActivateAction },
    { "deactivateAction",  DeactivateAction }
  };

  XtSetLanguageProc(NULL, (XtLanguageProc)NULL, NULL);

  XtAppAddActions(app_context, interface_actions, 6);


} // MarkAppInitialize()


/*********************************************/
/*      Translation Table()                  */
/*********************************************/

static char defaultTranslations[] =
    "<Btn2Down>:          btnDownAction()   \n\
     <Btn2Up>:            btnUpAction()     \n\
     <Btn2Motion>:        btnMotionAction() \n\
     Alt<Key>a:           activateAction() \n\
     Alt<Key>d:           deactivateAction() \n\
     Alt<Key>h:           helpAction()";


/*****************************************************************************/
/*        MarkInterface::MarkInterface()                                     */
/*****************************************************************************/

MarkInterface::MarkInterface(Widget w, char *classifier_file)
{
  _widget = w;
  _appCallbackProc = NULL;
  _userData = NULL;
  _interactionStatus = MARKING_ACTIVATED;

  _classifier = new MarkClassifier(classifier_file);
  if(!_widget) return;

  char *var;

  //    use default translations only if user doesn\'t define his
  if (!(var = getenv("MARKUSERTRANSLATIONS")) )
  {
    //  MODIFY WIDGET'S TRANSLATION TABLE
    XtTranslations widgetTranslations;
    widgetTranslations = XtParseTranslationTable(defaultTranslations);
    XtOverrideTranslations(_widget, widgetTranslations);
  }

  //    store pointer to the MarkInterface object in widget
  XtVaSetValues(_widget, XmNuserData, (XtPointer)this, NULL);

}/* end constructor */



/*****************************************************************************/
/*        MarkInterface::MarkInterface()                                     */
/*****************************************************************************/

MarkInterface::MarkInterface(Widget w, MarkClassifier *classifier)
{
  _widget = w;
  _appCallbackProc = NULL;
  _userData = NULL;
  _classifier = classifier;
  _interactionStatus = MARKING_ACTIVATED;

  if(!_widget) return;

  char *var;

  //    use default translations only if user doesn\'t define his
  if (!(var = getenv("MARKUSERTRANSLATIONS")) )
  {
    //  MODIFY WIDGET'S TRANSLATION TABLE
    XtTranslations widgetTranslations;
    widgetTranslations = XtParseTranslationTable(defaultTranslations);
    XtOverrideTranslations(_widget, widgetTranslations);
  }

  //    store pointer to the MarkInterface object in widget
  XtVaSetValues(_widget, XmNuserData, (XtPointer)this, NULL);

}/* end constructor */


/*****************************************************************************/
/*        initMark()                                                         */
/*****************************************************************************/

void
initMark(MarkEvent &mark_event)
{
  mark_event.widget = NULL;
  mark_event.start_time = mark_event.end_time = 0;
  mark_event.start_x = mark_event.start_y = mark_event.last_x = mark_event.last_y = 0;
  mark_event.state = 0;
  delete mark_event.path_points;  mark_event.path_points = NULL;
  mark_event.npoints = 0;
  delete mark_event.mark_class_name; mark_event.mark_class_name = NULL;

}/* initMark()
```

```
/*****************************************************************************/
/*          MarkInterface::eventHandler()                                    */
/*****************************************************************************/

void MarkInterface::eventHandler(Widget w, XEvent *event)
{
  static MarkPoint curp, lastp;
  static MarkEvent _mark_event;

  if(_interactionStatus == MARKING_DEACTIVATED) return;

  if(event->type == ButtonPress) {
    _curUserExample = new MarkExample;
    initMark(_mark_event);

    _mark_event.widget = w;
    curp.time = 0; _mark_event.start_time = event->xbutton.time;
    curp.x = _mark_event.start_x = event->xbutton.x;
    curp.y = _mark_event.start_y = event->xbutton.y;
    _curUserExample->add_point(curp);
  }

  else if(event->type == MotionNotify) {
    curp.time = 0;
    curp.x = event->xmotion.x;
    curp.y = event->xmotion.y;
    _curUserExample->add_point(curp);
  }

  else if (event->type == ButtonRelease) {
    curp.time = 0;
    curp.x = event->xbutton.x;
    curp.y = event->xbutton.y;
    _mark_event.end_time = event->xbutton.time;
    _curUserExample->add_point(curp);
    _mark_event.path_points = _curUserExample->points(&_mark_event.npoints);
    _mark_event.state = event->xbutton.state;
    _curUserExample->bbox(_mark_event.bbox);

    //  CLASSIFY THIS EXAMPLE
    if(_classifier) {
      const char *name = _classifier->classify_example(_curUserExample);
      if (name) {
        _mark_event.mark_class_name = new char[strlen(name) + 1];
        strcpy(_mark_event.mark_class_name, name);
      }
    }

    //  IF CLASSIFICATION SUCCESSFUL
    if(_mark_event.mark_class_name)
    {
      //        USER CALLBACK
      if(_appCallbackProc)
        (*_appCallbackProc)(this, _userData, &_mark_event);

      delete _curUserExample; _curUserExample = NULL;

    } // if classification successful

  } // buttonRelease

  else
    printf ("eventHandleer(): unrecognized event\n");

} // eventHandler()
```

```
/*****************************************************************************/
/*          MarkInterface::displayMarksRecognized()                          */
/*****************************************************************************/

void
MarkInterface::displayMarksRecognized()
{
  if (!_classifier) return;

  int nexamples = _classifier->nclasses();
  if (!nexamples) return;

  //    SET UP THE WIDGET TO DISPLAY MARKS

  Widget shell_w, da_w, pane_w, label_w, pushb_w, rowcol_w, drawb_w;
  Widget scroll_w;
  GC da_gc;

  shell_w = XtVaCreatePopupShell("shell_help", xmDialogShellWidgetClass,
                                 get_top_shell(_widget),
                                 XmNdeleteResponse, XmDESTROY,
                                 NULL);

  pane_w = XtVaCreateWidget("pane_help", xmPanedWindowWidgetClass, shell_w,
                            XmNsashWidth, 1,
                            XmNsashHeight, 1,
                            NULL);

  //    TOP LABEL

  char *widget_name = (char *) XtName(_widget);
  char help_message[] = "Marks Recognized by Widget: ";
  char *label_message = new char [strlen(widget_name) + strlen(help_message) + 2];
  strcpy(label_message, help_message);
  strcat(label_message, widget_name);
  XmString label_str = XmStringCreateSimple(label_message);

  label_w = XtVaCreateManagedWidget("help_label",
                                    xmLabelWidgetClass, pane_w,
                                    XmNlabelString, label_str,
                                    NULL);

  XmStringFree(label_str);
  delete label_message;

  //    SCROLLING WINDOW WITH DRAWN BUTTONS FOR REP-EXAMPLES

  int nrows = (nexamples/EXAMPLES_PER_ROW) + 1;
  int ncols = (nexamples >= EXAMPLES_PER_ROW) ? EXAMPLES_PER_ROW : nexamples;
  int sw_width = (EXAMPLE_WIDTH + SCROLLW_OFFSET) * ncols;  // no scrolling along x-axis
  int sw_height = (EXAMPLE_HEIGHT + SCROLLW_OFFSET) * ( (nrows < EXAMPLES_PER_COL) ? nrow
s : EXAMPLES_PER_COL);

  scroll_w = XtVaCreateManagedWidget("scrolledw_help",
                                     xmScrolledWindowWidgetClass, pane_w,
                                     XmNwidth, (sw_width),
                                     XmNheight, (sw_height),
                                     XmNscrollingPolicy, XmAUTOMATIC,
                                     NULL);

  rowcol_w = XtVaCreateManagedWidget("rowcol_help", xmRowColumnWidgetClass,
                                     scroll_w,
                                     XmNpacking,  XmPACK_COLUMN,
                                     XmNnumColumns, nrows,
```

```
                                XmNorientation, XmHORIZONTAL,
                                NULL);



  int example_index = 0;
  for (int i = 0; i < nrows; ++i) {
    for (int j = 0; j < ncols; ++j) {
      if (example_index == nexamples) break;
      drawb_w = XtVaCreateManagedWidget("drawbutton_help", xmDrawnButtonWidgetClass,
                                        rowcol_w,
                                        // XmNlabelType, XmPIXMAP,
                                        XmNwidth, EXAMPLE_WIDTH,
                                        XmNheight, EXAMPLE_HEIGHT,
                                        XmNpushButtonEnabled, True,
                                        XmNshadowType, XmSHADOW_ETCHED_OUT,
                                        NULL);
      XtAddCallback(drawb_w, XmNexposeCallback, drawb_callback, (void *)example_index);
      XtAddCallback(drawb_w, XmNactivateCallback, drawb_callback, (void *)example_index)
;
      //        store pointer to the MarkInterface object in widget
      XtVaSetValues(drawb_w, XmNuserData, (XtPointer)this, NULL);
      example_index++;
    }
    if (example_index == nexamples) break;
  }

  //    MESSAGE BUTTON
  XmString msg_str = XmStringCreateSimple("Click on a Button to Activate/Deactivate Mark
");

  label_w = XtVaCreateManagedWidget("help_label",
                                    xmLabelWidgetClass, pane_w,
                                    XmNlabelString, msg_str,
                                    NULL);

  //    CANCEL BUTTON

  label_str = XmStringCreateSimple("Cancel");

  pushb_w = XtVaCreateManagedWidget("cancel",
                                    xmPushButtonWidgetClass, pane_w,
                                    XmNlabelString, label_str,
                                    NULL);

  XmStringFree(msg_str);
  XmStringFree(label_str);

  XtAddCallback(pushb_w, XmNactivateCallback, dismiss_shell_callback, shell_w);

  XtManageChild(rowcol_w);
  XtManageChild(pane_w);

  XtPopup(shell_w, XtGrabNone);


} /* displayMarksRecognized() */



/*****************************************************************/
/* get_top_shell()                                               */
/*****************************************************************/

Widget
get_top_shell(Widget w)
{
  while (w && !XtIsWMShell(w))
    w = XtParent(w);
  return w;
}


/*****************************************************************
* HANDLE : dismiss_callback
*
* callback for inputs to drawing area
*****************************************************************/

static void
dismiss_shell_callback(Widget w, XtPointer shell, void *cbs)
{
  XtDestroyWidget(Widget(shell));

}


/*****************************************************************
* HANDLE : drawb_callback
*
* callback from the draw-buttons
*****************************************************************/

static void
drawb_callback(Widget drawb_w, XtPointer client_data, void *cb)
{
  XmDrawnButtonCallbackStruct *cbs = (XmDrawnButtonCallbackStruct *)cb;
  int example_index = (int) client_data;
  MarkInterface *mi;
  XtVaGetValues(drawb_w, XmNuserData, &mi, NULL);

  Display *drawb_disp = XtDisplay(drawb_w);
  Screen *drawb_screen = XtScreen(drawb_w);

  //    use the same gc for all the draw-buttons
  static GC drawb_enabled_gc;
  static GC drawb_disabled_gc;
  static XFontStruct *font_struct;
  static char        *font_name;

  if (!drawb_enabled_gc) {
    drawb_enabled_gc = XCreateGC(drawb_disp, RootWindowOfScreen(drawb_screen),
                                 0, NULL);
    drawb_disabled_gc = XCreateGC(drawb_disp, RootWindowOfScreen(drawb_screen),
                                  0, NULL);

    //  GET COLOR PIXEL VALUES FROM COLOR NAMES
    Pixel enable_fg, enable_bg, disable_fg, disable_bg;
    XColor enable_xcolor_fg, enable_xcolor_bg, unused;
    Colormap cmap = DefaultColormapOfScreen(drawb_screen);

    XAllocNamedColor(drawb_disp, cmap, ENABLE_FG, &enable_xcolor_fg, &unused);
    XAllocNamedColor(drawb_disp, cmap, ENABLE_BG, &enable_xcolor_bg, &unused);
    enable_fg = enable_xcolor_fg.pixel;
    enable_bg = enable_xcolor_bg.pixel;
    disable_fg = enable_bg;
    disable_bg = enable_fg;

    XSetBackground(drawb_disp, drawb_enabled_gc, enable_bg);
    XSetForeground(drawb_disp, drawb_enabled_gc, enable_fg);

    XSetBackground(drawb_disp, drawb_disabled_gc, disable_bg);
    XSetForeground(drawb_disp, drawb_disabled_gc, disable_fg);
```

```
    font_name = "-adobe-times-bold-r-normal--14-140-75-75-p-77-iso8859-1";
    font_struct = XLoadQueryFont(drawb_disp, font_name);
    if(font_struct == NULL)
      printf("font_struct = null\n");

  }

  Window drawb_win = cbs->window;

  // XFillRectangle(drawb_disp, drawb_win, drawb_gc, 0, 0,
  //    EXAMPLE_WIDTH, EXAMPLE_HEIGHT);

  XEvent *db_event = cbs->event;

  if (cbs->reason == XmCR_EXPOSE) {
    if(mi->enabled(example_index) == RECOG_ENABLED) {
      mi->drawExample(drawb_w, drawb_win, drawb_enabled_gc, example_index);
    }
    else {
      mi->drawExample(drawb_w, drawb_win, drawb_disabled_gc, example_index);
    }
  }

  else if(cbs->reason == XmCR_ACTIVATE)
  {
    mi->toggleRecognitionStatus(example_index);
    if(mi->enabled(example_index)) {
      mi->drawExample(drawb_w, drawb_win, drawb_enabled_gc, example_index);
    }
    else {
      mi->drawExample(drawb_w, drawb_win, drawb_disabled_gc, example_index);
    }
  }

} /* drawb_callback() */



/*****************************************************************/
/*              MarkInterface::drawExample()                     */
/*****************************************************************/

void
MarkInterface::drawExample(Widget db_w, Drawable db_win, GC db_gc,
                           int example_index)
{
  if (example_index < 0) return;

  const MarkExample *ex = _classifier->rep_example(example_index);
  if (!ex || ex->empty()) return;


  Display *db_display = XtDisplay(db_w);
  unsigned int npoints;
  register int i;
  register MarkPoint p1, p2;

  MarkPoint *points = ex->points(&npoints);
  MarkBBox bbox;
  ex->bbox(bbox);
  double max_x = bbox.max_x;
  double max_y = bbox.max_y;
  double min_x = bbox.min_x;
  double min_y = bbox.min_y;

  double mag_x = (EXAMPLE_WIDTH/(max_x));
  double mag_y = (EXAMPLE_HEIGHT/(max_y));

  //    CREATE THE MAGNIFIED(COMPRESSED) EXAMPLE
  MarkExample mag_ex;

  for (i = 0; i < (npoints - 1); ++i)
  {
    p1.x = (mag_x) * (points[i].x);
    p1.y = (mag_y) * (points[i].y);
    p1.time = 0;
    mag_ex.add_point(p1);
  }

  delete [] points;

  //    CENTER THE NEW EXAMPLE IN THE DRAWING BUTTON

  int cx = EXAMPLE_WIDTH/2;
  int cy = EXAMPLE_HEIGHT/2;

  mag_ex.bbox(bbox);    // bbox of the new magnified example

  int cdx = cx - ((bbox.max_x + bbox.min_x)/2);
  int cdy = cy - ((bbox.max_y + bbox.min_y)/2);

  points = mag_ex.points(&npoints);

  for (i = 0; i < (npoints - 1); ++i)
  {
    int x1 = points[i].x + cdx;
    int y1 = points[i].y + cdy;
    int x2 = points[i + 1].x + cdx;
    int y2 = points[i + 1].y + cdy;
    XDrawLine(db_display, db_win, db_gc, x1, y1, x2, y2);
  }

  delete [] points;

  const char *name = ex->name();
  if (name) {
    int str_len = strlen(name);
    XDrawString(db_display, db_win, db_gc, NAME_START_X, NAME_START_Y,
                name, str_len);
  }

} /* drawExample() */


/*****************************************************************/
/*              MarkInterface::toggleRecognitionStatus()         */
/*****************************************************************/

void
MarkInterface::toggleRecognitionStatus(const char *mark_name) const
{
 if(_classifier)
   _classifier->toggle_recognition_status(mark_name);
}

/*****************************************************************/
/*              MarkInterface::toggleRecognitionStatus()         */
/*****************************************************************/

void
MarkInterface::toggleRecognitionStatus(int mark_index) const
{
  if(_classifier)
```

```
    _classifier->toggle_recognition_status(mark_index);
}

/*****************************************************************/
/*              MarkInterface::enabled()                         */
/*****************************************************************/

int
MarkInterface::enabled(const char *mark_name) const
{
  if(_classifier)
    return (int) _classifier->enabled(mark_name);
  else return RECOG_DISABLED;

}

/*****************************************************************/
/*              MarkInterface::enabled()                         */
/*****************************************************************/

int
MarkInterface::enabled(int mark_index) const
{
  if(_classifier)
    return (int) _classifier->enabled(mark_index);
  else return RECOG_DISABLED;

}

/*****************************************************************/
/*              MarkInterface::get_widget()                      */
/*****************************************************************/

Widget
MarkInterface::get_widget() const
{
  return _widget;
}


/*****************************************************************/
/*              MarkInterface::get_classifier()                  */
/*****************************************************************/

const MarkClassifier*
MarkInterface::get_classifier() const
{
  return _classifier;
}

/*****************************************************************/
/*              MarkInterface::get_callbackProc()                */
/*****************************************************************/

MarkCallbackProc
MarkInterface::get_callbackProc() const
{
  return _appCallbackProc;

}

/*****************************************************************/
/*              MarkInterface::set_callbackProc()                */
/*****************************************************************/

void
MarkInterface::set_callbackProc(MarkCallbackProc new_callback,
                                void *new_data)
{
  _appCallbackProc = new_callback;
  _userData = new_data;

}

/*****************************************************************/
/*              MarkInterface::get_userData()                    */
/*****************************************************************/

void *
MarkInterface::get_userData() const
{
  return _userData;
}

/*****************************************************************/
/*              MarkInterface::set_userData()                    */
/*****************************************************************/

void
MarkInterface::set_userData(void *new_data)
{
  _userData = new_data;
}


/*****************************************************************/
/*              MarkInterface::activateMarkingInteraction()      */
/*****************************************************************/

void
MarkInterface::activateMarkingInteraction()
{
  _interactionStatus = MARKING_ACTIVATED;
}

/*****************************************************************/
/*              MarkInterface::deactivateMarkingInteraction()    */
/*****************************************************************/

void
MarkInterface::deactivateMarkingInteraction()
{
  _interactionStatus = MARKING_DEACTIVATED;
}
```

```
/*****************************************************************/
/* example.C                                                     */
/*                                                               */
/* Author: Lalit K. Agarwal, Brown University                    */
/*                                                               */
/* Date:   May 1994                                              */
/*                                                               */
/* An example program to show how to use a MarkInterface object  */
/* to augment a Motif widget with marking interaction.           */
/*****************************************************************/


/*************************************************************/
/* General i/o include files                                 */
/*************************************************************/

#include <stdio.h>
#include <stdlib.h>

/*************************************************************/
/* Include files required for all Toolkit programs           */
/*************************************************************/

#include <X11/Intrinsic.h>      /* Intrinsics Definitions */
#include <X11/StringDefs.h>     /* Standard Name-String definitions */

/*************************************************************/
/* Include files for widgets actually used in this appln.    */
/*************************************************************/

#include <Xm/MainW.h>
#include <Xm/DrawingA.h>
#include <Xm/PanedW.h>
#include <Xm/PushB.h>
#include <Xm/RowColumn.h>

/*************************************************************/
/* Local include files for classes                           */
/*************************************************************/

#include "MarkInterface.H"

/***********************************************/
/*             app_gesture_handler1()          */
/***********************************************/

void app_gesture_handler1(MarkInterface *mi, void *user_data,
                          MarkEvent *mark_event)
{
  if (mark_event->mark_class_name)
    printf("mark = %s\n", mark_event->mark_class_name);
}


/***********************************************/
/*             app_gesture_handler2()          */
/***********************************************/

void app_gesture_handler2(MarkInterface *mi, void *user_data,
                          MarkEvent *mark_event)
{
  if (mark_event->mark_class_name)
    printf("mark = %s\n", mark_event->mark_class_name);
}

/***********************************************/
/*             main()                          */
/***********************************************/

main(int argc, char **argv)
{
  XtAppContext app_context;
  Widget toplevel;
  Widget main_w;
  Widget da_w1, da_w2;
  Widget pane_w;

  toplevel = XtVaAppInitialize(
                                  &app_context,          /* Application context */
                                  "MarkEv",              /* Application class, name of reso
urce file */
                                  NULL, 0,               /* command line option list */
                                  &argc, argv,           /* command line args */
                                  NULL,                  /* for missing app-defaults file*/
                                  NULL);


  main_w =  XtVaCreateManagedWidget("MainWindow",
                                    xmMainWindowWidgetClass,
                                    toplevel,
                                    NULL);

  pane_w = XtVaCreateWidget("pane_w", xmPanedWindowWidgetClass, main_w,
                            XmNsashWidth, 1,
                            XmNsashHeight, 1,
                            NULL);

  da_w1 = XtVaCreateWidget ("drawing_area1",
                            xmDrawingAreaWidgetClass, pane_w,
                            XmNwidth, 400,
                            XmNheight, 200,
                            NULL);


  da_w2 = XtVaCreateWidget ("drawing_area2",
                            xmDrawingAreaWidgetClass, pane_w,
                            XmNwidth, 400,
                            XmNheight, 200,
                            NULL);

  XtManageChild(da_w2);
  XtManageChild(da_w1);
  XtManageChild(pane_w);

  //    Initiliaze the Marks package
  MarkAppInitialize(app_context);

  //    Create a MarkInterface object for each drawing area
  //    and link it to the widget.

  MarkInterface mi2(da_w2, argv[2]);
  MarkInterface mi1(da_w1, argv[1]);

  //    Register the mark-related callbacks with the MarkInterface objects.
  mi1.set_callbackProc(app_gesture_handler1);
  mi2.set_callbackProc(app_gesture_handler2);

  XtRealizeWidget(toplevel);
  XtAppMainLoop(app_context);

  return 1;

} // main()
```