Chapter 14
Creating New Model Objects

This chapter is designed to guide those who are programming new model objects for ViSta (version B 1.0r1, Oct. 1992). While the chapter has not been updated to correspond to newer releases, the general outline and most of the details still apply. We discuss ViSta’s data and model object systems, and how to develop new ViSta model objects. This chapter assumes the developer is familiar with Lisp-Stat (Tierney, 1990).
14.1 Data Objects

Data objects are used to define data used by ViSta. In Chapter 3 we discussed the function which defines a data object, and the keywords which can be used with the function. In this section we discuss the data object prototype inheritance structure, the concept of the current data object, and we present messages which can be send to a data object which has already been defined.

14.1.1 Inheritance Structure

Note that there are three prototype data objects, one for multivariate data, one for dissimilarity data, and one for tabular data. The multivariate data object is named \texttt{mv-data-object-proto}; the dissimilarity data object, which inherits from \texttt{mv-data-object-proto} is named \texttt{diss-data-object-proto}; the tabular data object, which inherits from \texttt{mv-data-object-proto}, is named \texttt{table-data-object-proto}. The \texttt{data} function creates a data object which is an instance of one of these three data object prototypes. If the \texttt{:matrices}, and \texttt{:classes} keywords are not used in the \texttt{data} function when the data object is defined, then the data object is a multivariate data object and is an instance of \texttt{mv-data-object-proto}. If the \texttt{:matrices} keyword is used in the \texttt{data} function when the data object is defined, then the data object is a dissimilarity data object and is an instance of \texttt{diss-data-object-proto}. If the \texttt{:classes} keyword is used in the \texttt{data} function when the data object is defined, then the data object is a tabular data object and is an instance of \texttt{table-data-object-proto}.

14.1.2 Current Data and Data Object Names

One of the data objects is always the “current” data. The current data is indicated in the data menu by the checked menu item. It is also indicated in the workmap as the highlighted data icon. The data’s object identification can be found in the global variable \texttt{current-data}, to which messages can always be sent. The current data can be changed by clicking on a new data icon, by choosing a new item of the data menu.

When a data object is defined by the \texttt{data} function, the function also defines a variable whose name is the name of the data object, and whose value is the object’s identification information. Using this name, the current data can be changed from the keyboard by using the \texttt{setcd} (set current-data) function. For example, if there is a data object named \texttt{crime}, you can make it the current data by typing \texttt{(setcd crime)}.
14.1.3 Data Object Messages and Methods

Once a data object has been created it can be sent messages. The messages can be sent either to current-data or to a specific data object by using the name of the data object. For example, the message (send current-data :data) returns the list of data values for the current-data. If there is a data object named crime, you could type (send crime :data) to see a listing of its data values.

There is a group of messages which can be used to either obtain information about the data object or to store information in the data object. You obtain information about the object by using the message without an argument, while you can store information in the object by following the message with an argument that is the information. For example (send crime :title) causes the data-object to tell you its title, whereas (send crime :title “Crime 1992”) allows you to send a title to the object. In addition to the :title message, messages in this group are :nobs, which returns or sets the number of observations in the data object; :nvar, which returns or sets the number of variables in the object; :variables, which returns or sets the names of the variables; :labels, which returns or sets the labels of observations; :menu-length, which returns or sets the position of the object’s name in the data menu; :types, which returns or sets the type of each variable; :data, which returns or sets the data-object data list; and :name which returns or sets the name of the data object. Also, :obs-states, :var-states and :mat-states return or set the state of each observation, variable or matrix (the states can be normal, selected or invisible); :obs-window, :var-window and :mat-window return or set a logical value of t or nil indicating whether or not the window that lists observations, variables or matrices should be open when the data object is the current object; and :obs-window-object, :var-window-object and :mat-window-object return or set the object identification information for each window. The message :matrices can be used with dissimilarity data to obtain the names of matrices for dissimilarity data. With the optional list of character strings, the message specifies that the data are dissimilarity data and specifies the number the matrices and their names.

There is another group of messages which are sent to the current-data by the menu items in the data menu and in the workmap and data icon popup menus. Each message corresponds to the menu item having the same name. For example, the message (send current-data :save-data) is used by the save data menu item of the data menu and workmap popup menu. Each message has a short form: (save-data) is the short form of the message given above. By using the long form, the programmer may also send these messages to a specifically named data object. The messages include :save-data, which saves data into a ViSta datafile specified in a dialog-box (the message can be followed by a string argument to name the file, if desired); :create-data, which creates a new data object from the active portion of the current data; :visualize to see a visualization of the
data; :report to see a report (listing) of the data; :datasheet to obtain a datasheet containing the data (this message is not yet implemented); :list-variables which lists the variables in the current data (you can also use :list-vars or :list-var); :list-observations (or :list-obs) which lists the observations in the current data; :list-matrices (or :list-mats or :list-mat) which lists the matrices in dissimilarity data; :merge-variables, which merges the active variables in the current data with the active variables in the previously current data; and :merge-observations, which merges the active observations in the current data with the active observations in the previously current data. The last two messages take an optional argument which is a character string that is used to name the new data object. If no character string is present, a dialog box is presented to obtain the name. Finally, you can type :summary to see a summary report (listing of summary statistics) of the data. The :summary message has five keyword arguments, each of which must be followed with t or nil (the default). These arguments include :moments, :quartiles, :ranges, and :covariances, which determine what type of summary statistics are reported, and :dialog, which determines whether or not a dialog box is presented to obtain the desired types of statistics.

There is a group of messages which take an argument that is a list that specifies the "type" of the variable. The elements of the list must be one of the character strings "labels" "category" "ordinal" "numeric" or "all" (in upper-case, lower-case or mixed-case). These messages return information about "active" variables (variables which appear in the "Variables" window and are selected, if any are selected) whose types match the specified value of ok-types. The messages are :active-data (to get the active portion of the data) :active-data-matrix (to get the active portion of the data in matrix form), :active-nvar (to find out how many active variables there are) :active-variables (to obtain a list of the names of the active variables) :active-types (to obtain a list of the types of the active variables), and :active-matrices (to obtain a list of the names of the active matrices for dissimilarity data).

Other data object messages include :variable which reports a list of values for the variable whose name is specified by the string argument of the message. Various simple statistics of the active variables in the data can be obtained with the messages :covariance-matrix, :interquartile-ranges, :kurtoses, :maximums, :means, :medians, :mid-ranges, :minimums, :ranges, :skewnesses, :standard-deviations, and :variances. None of these messages take an argument. The message :close-windows closes any windows that are open for the current object.
14.2 Model Objects

There are three classes of prototype model objects, each of which inherits from a prototype data object, as shown in Figure 2. One class of prototype model objects inherits from `mv-model-object,proto` which in turn inherits from `mv-data-object,proto`. The second class of prototype model objects inherit from `tab-model-objectPROTO`, which itself inherits from the `tab-data-object,proto`, which in turn inherits from the `mv-data-objectPROTO`. The third class of prototype model objects inherit from the `diss-model-objectPROTO` which in turn inherits from the `diss-data-objectPROTO`. Within each of these classes of prototype model objects there are prototype model objects that are specialized to a specific analysis method that is appropriate for the particular type of data from which the prototype objects inherit.

The prototype model objects which inherit from `mv-model-objectPROTO` are for analysis methods that are appropriate to multivariate data. Currently, these model object prototypes include `standardize-object-proto` (for standardizing multivariate data), `pca-model-object-proto` (for principal components), `mmr-model-object-proto` (for multivariate multiple regression) and `crs-model-object-proto` (for correspondence analysis). The prototype model objects which inherit from `tab-model-object-proto` are appropriate for analysis of data tables. Currently, there is no tabular model objects, but in the future the `anova-model-object-proto` (for analysis of variance) and `discrim-model-object-proto` (for discriminant analysis) will appear here in the hierarchy. The model objects which inherit from `diss-model-object-proto` are for analysis methods which are appropriate to dissimilarity data. Currently there is

---

**Figure 2: Data and Model Object Inheritance Structure**

![Diagram showing the inheritance structure of data and model objects.](image-url)
one such model object prototype, mds-model-object-proto for multidimensional scaling\(^1\). In the future, cls-model-object-proto (for cluster analysis) will appear here. Since every model prototype ultimately inherits from the multivariate data prototype, all messages and methods given in Section 14.1 can be used with any model.

### 14.2.1 Current Model and Model Object Names

One of the models is always the “current” model. The model’s object identification is in the global variable current-model, to which messages can always be sent. The current model is indicated in the model menu by the checked menu item, and in the workmap by the high-lighted model icon. The current model can be changed by clicking on a new model icon or choosing a new menu item in the model menu. Every model has a name which appears below the model’s icon and in the model menu. The name can be used in the setcm function to change the current model from the keyboard. If there is a model named pca-crime, you can type (setcm pca-crime) to change the current model to pca-crime. You can also send messages to any model by typing, for example (send pca-crime :title).

### 14.2.2 Multivariate Model Object Messages and Methods

In this section we overview the sequence of actions that occurs when the user creates a new instance of a model, noting the methods that are used to create the instance. We then we outline the methods that every model prototype must have, so that you will know how to write a new model prototype.\(^2\)

All multivariate model objects have many methods. Since model prototypes inherit from data prototypes, the model objects inherit all of the data object methods. In addition, there are some methods which are shared by all multivariate model objects. These methods have already been written as methods of mv-model-object-proto, and, due to the inheritance structure, are inherited by all model prototypes. However, some of the methods that are shared by all multivariate model objects must be uniquely redefined for individual model objects so that their actions are appropriate to the particular model. You will have to write these methods.

---

1. Currently, the multidimensional scaling object does not inherit from other objects as described here, nor does its code structure conform to the code structure given in this section.
2. To understand this and the next section, you should look at the code for one or more model objects as you read the section. The correspondence analysis or principal components model objects may be the best to follow. In addition, the code for mv-model-object-proto should be consulted (see the file modelobj.lsp). The code structure of the multidimensional scaling model object does not correspond to the description given in this section. It should not be reviewed at this point.
Finally, some methods are unique to an individual model object. You will also have to be write these methods.

When the user requests a data analysis (by using, for example, a menu item of the method menu) a new instance of a model prototype is created by the model’s constructor function. This function is used by the method menu and can be typed at the keyboard by the user. As an example, the constructor function for principal components analysis is simply (principal-components). Constructor functions usually take arguments. For example (principal-components :covariances t), when the user wishes to analyze covariances. As another example, the principal components menu item issues the statement (principal-components :dialog t), so that a dialog box is presented to obtain the argument values.

The model’s constructor function first checks on the validity of the argument values, and then issues the :new message followed by a set of arguments. The arguments must be as follows: The first several arguments are the parameters specific to the particular model (such as covariances for the principal components model). Then, there are five arguments: 1) an integer which specifies the number of the model’s method-button; 2) the data list for the analysis; 3) the model’s title; 4) the model’s name; and 5) a logical value indicating whether a dialog box is to be presented to obtain parameter values.

The constructor function’s :new message invokes the model’s :isnew method. This method creates a new instance of the model, saves the parameters of the model in the instance’s slots, and then uses call-next-method to call the mv-model-object-proto’s :isnew method. This method takes a series of actions: (1) the mv-data-object-proto’s :isnew method is called by call-next-method with arguments that are the model’s data, variable list, title, observation labels, variable types and object name; (2) the model’s method icon is added to the workmap; (3) the model’s option dialog box is presented; then, if the dialog is successful, (4) the object identification information about the data that are being analyzed is saved in a slot of the model; (5) the analysis takes place; (6) any open data object windows are closed; and (7) the model icon is added to the workmap.

The mv-model-object-proto’s :isnew method takes the actions outlined in the previous paragraph by sending messages to various objects. The messages, in order, are :copy-tool-icon (sent to the toolbox), :dialog, :options, :data-object, :analysis, (all sent to the model), :close-windows (send to the data), and :new-model (sent to the model). Note that when dialog is t and when :options returns a value of nil (for example, when the option dialog’s cancel button is pressed, or when the dialog is incorrectly used) the remaining messages are not sent.
The :dialog method sets the model’s dialog slot to t or nil, depending on the value of dialog, whereas the :options method presents the model’s options dialog. The :data-object method save the object identification information about the data that are being analyzed in a model slot. The :analysis method performs the analysis. Once the analysis has successfully completed, the :new-model method updates the system to recognize the new model object. This method assigns the current-model and current-object global variables a value equal to the object identifier of the model and whose value is the model’s object identification information; append a new item to the bottom of the model menu; and create an action for the menu item which is to perform the setcm function. Thus, whenever the new menu item is chosen (and, indirectly, whenever the model icon is selected), the actions contained in setcm take place.

All models have the ability to create a copy of the data that was input to them. This is done by the :create-input-data-object method which creates a new data object that contains the data input to the model. Finally, the abbreviation for the model can be obtained from the model by the message (send current-model :model-abbrev). This message takes an optional argument that sets the model abbreviation.

All models must have six additional methods. The specific details of these methods depends on the specific model - each of the methods must be uniquely defined for each model, thus, when you develop a new model object you must define methods for each of these messages. Two of these methods have already been discussed above. These are the :options method, which presents a dialog that allows setting options for models, and the :analysis method, which performs the analysis that fits the model to the current-data. The other methods are used by the model menu. These are :save-model-template, which saves a copy of the model; :create-data, which creates one or more data objects of the results of the model; :visualize, which returns a visualization of the model; and :report, which returns a report of the results of the analysis. In addition to these methods, you may need to define additional methods that are used by specific models.

14.2.3 Control-Flow when Model Objects are Constructed

(These are notes that need to be added somewhere in this chapter - FWY).

The default (or user specified) values in the model constructor function are put into the model slots by the :isnew method. Then the :options method is called by the ancestors of your object (you don’t call it yourself). If :dialog is true, the :options method is used to show the dialog to get values for the options, which may or may
not be different than those gotten from the constructor function. The :options method then puts these values, whatever they are, in the slots, ignoring any values that may already be in them.

The :options method returns nil when it is not used right, or when it is canceled. Otherwise, it returns the list of dialog item values, tho this list isn't used. If the dialog is used correctly, the :options method should stuff the values gotten from the dialog items into the appropriate slots.

Then the ancestors of your method determine when the :analysis method should be used (depending on the outcome of the :options method). If so, the :analysis method gets called by the ancestors of your object, no matter what :dialog is. The :analysis method then uses whatever is in the slots.

The :analysis method doesn't use the list of dialog results (in fact, nobody uses this list), rather, it uses the information that has been put into the slots. More generally, no method uses the results of any other method directly. All communication between methods is done via slots.
14.3 An Example: The Principal Components Model Object

In this section we show the steps taken to write a model object for Principal Components model analysis. This example can be followed by the developer who wishes to develop a new model object, as the steps are the same for all objects. The code shown here is from the `PCAMOB.LSP` file.

**Step 1: Define the model prototype:** The first step in defining a new model object is to define the model’s prototype object. The `defproto` statement for the Principal Components model object is:

```
(defproto pca-model-object-proto
  '(scores coefs eigenvalues svd corr) ()
  mv-model-object-proto)
```

The slots `scores coefs eigenvalues svd` and `corr` will hold information specific to this model. The prototype inherits from `mv-model-object-proto`.

**Step 2: Define slot-accessor methods.** There must be a slot accessor method for each slot specified in the `defproto` function. For principal components these methods are:

```
(defmeth pca-model-object-proto :scores
  (optional (values nil set))
  (if set (setf (slot-value 'scores) values))
  (slot-value 'scores))

(defmeth pca-model-object-proto :coefs
  (optional (values nil set))
  (if set (setf (slot-value 'coefs) values))
  (slot-value 'coefs))

(defmeth pca-model-object-proto :eigenvalues
  (optional (values nil set))
  (if set (setf (slot-value 'eigenvalues) values))
  (slot-value 'eigenvalues))

(defmeth pca-model-object-proto :svd
  (optional (structure nil set))
  (if set (setf (slot-value 'svd) structure))
  (slot-value 'svd))

(defmeth pca-model-object-proto :corr
  (optional (val nil set))
  (if set (setf (slot-value 'corr) val))
  (slot-value 'corr))
```
The :scores and :coefs slots will contain matrices, and the :eigenvalues a vector of the principal component scores, coefficients and eigenvalues. The :svd slot will contain the structure resulting from the sv-decomp function (which is redundant with the information in the first three slots, but which facilitates retrieving the information). The corr slot will contain t or nil to indicate whether correlations or covariances are to be computed from the multivariate data.

**Step 3: Define ViSta system methods.** As indicated in section 14.2.2, each model prototype must contain certain methods for ViSta to work properly. These methods must be named options and analysis (which are used by the isnew method of mv-model-object-proto), and save-model-template, create-data, report and visualize (all used by the menu system). We discuss these methods here, and indicate how similar methods would be defined for other model objects.

**Options:** The options method for the principal components proto is:

```lisp
(defmeth pca-model-object-proto :options ()
  (when (send self :dialog)
    (let ((result nil)
          (dialog-value (choose-item-dialog
                         "Analysis Options:"
                         '("Analyze Covariances"
                           "Analyze Correlations") :initial 1)))
      (when dialog-value
        (when (= 1 dialog-value) (def result t))
        (send self :corr result))
      dialog-value)))
```

This method first checks the value of the dialog slot to see if a dialog is to be presented. This action should always be taken first since when a script file job is in progress, or when a model is being loaded, the dialog should not be presented. The model’s constructor function should set the value of dialog properly, as it does in this example. When appropriate, the dialog is then presented to see if the analysis is to be performed on correlations or covariances. The dialog sets the value of the corr slot for later use by the :analysis method. Note that the results of the dialog are communicated to the analysis method through the slot. This should always be this way for all model prototypes. Then, the value that is returned by choose-item-dialog (which is nil when the dialog is canceled) is used as the value returned by :options, so that the analysis can be canceled when the dialog has been canceled. The :options method should always return nil in this situation for all models. Indeed, it should return nil whenever the options dialog is not properly used (see the multivariate multiple regression options dialog for a good example of this).
Analysis: The analysis method for the principal components proto is:

```
(defmeth pca-model-object-proto :analysis ()
  (let* ((left-alpha 1)
         (data (send self :data-matrix))
         (prepped-data
          (if (send self :corr)
              (/ (normalize (center data) 1)
                  (sqrt (1- (select (array-dimensions data) 0))))
              (/ (center data)
                  (sqrt (1- (select (array-dimensions data) 0)))))
          )
          (svd (sv-decomp2 prepped-data))
          (svd (if (< (sum (col (select svd 2) 0)) 0)
                  (list (* -1 (select svd 0))
                        (select svd 1)
                        (* -1 (select svd 2))
                        (select svd 3))
                  svd))
          (scores (matmult (select svd 0)
                           (diagonal (* (select svd 1) left-alpha))))
          (eigenvalues (* (select svd 1) 2))
          (coefs (matmult (select svd 2)
                           (diagonal (* (select svd 1)
                                        (1- left-alpha)))))))
    (send self :svd svd)
    (send self :coefs coefs)
    (send self :scores scores)
    (send self :eigenvalues eigenvalues)))
```

The details of this method are not important for this example, other than to note that all values (i.e., svd, coefs, scores and eigenvalues) are computed inside a let* statement (so that they are only locally defined) and then saved for later use by placing them in appropriate slots.

Save-Model-Template: Every model object needs to have a method named save-model-template, a method which is used by the model menu's save-model method. The method contains a template of the code that creates the model object. The method must always have as an argument the object identification information for the data object used by the model, since the data must be saved along with the model. For principal components, the method is:

```
ViSta: The Visual Statistics System
```
```lisp
(defmeth pca-model-object-proto :save-model-template
  (data-object)
  `(principal-components
     :title       ,(send self :title)
     :name        ,(send self :name)
     :dialog      nil
     :covariances ,(not (send self :corr))
     :data (data  ,(send data-object :name)
               :title     ,(send data-object :title)
               :variables ',(send self :variables)
               :types     ',(send self :types)
               :labels    ',(send self :labels)
               :data      ',(send self :data))))

Note the unusual backquote syntax (which is explained briefly by Tierney on pp. 98 and 120, as well as on page 197 in the discussion of saving objects). To be clear, the character in front of `(principal-components is a backquote. This character in front of a list causes all elements of the list to be quoted, except those preceded by commas, which are treated in the normal fashion. Note that the data for the principal components function come from the model object via the various (send self functions. However, the data’s name and title must come from the original data object that was analyzed.

The mv-model-object-proto’s :save-model method takes the quoted function and places it in a file, with the functions following commas being replaced with their results. When the file is loaded back into ViSta by (load-data), a principal components analysis is performed on the data that were also saved in the file.

ViSta: The Visual Statistics System
Create-Data: For principal components, just as for any other model object, the method which creates output data objects must work in such a way that it can be used by the menus, can be typed at the keyboard, or can be contained in a script file. The method is:

```lisp
(defun pca-model-object-proto :create-data
  (&key (dialog nil) (scores t) (coefs t) (input nil))
  (if (not (eq current-object self)) (setcm self))
  (let ((creator (send *desktop* :selected-icon))
        (desires (list (list (if scores 0)
                          (if coefs 1)
                          (if input 2))))
    (if dialog
      (setf desires
        (choose-subset-dialog
          "Choose Desired Data Objects"
          '("Component Scores"
             "Component Coefficients"
             "Analyzed Input Data")
          :initial (select desires 0)))))
    (when desires
      (when (member '0 (select desires 0))
        (send current-model
          :pca-scores-data-object creator))
      (when (member '1 (select desires 0))
        (send current-model
          :pca-coefs-data-object creator))
      (when (member '2 (select desires 0))
        (send current-model
          :create-input-data-object "PCA" creator)))
    )
)
```

First, note the statement

```
(if (not (eq current-object self))
  (setcm self))
```

This statement must be used at the beginning of the method to assure that the current model icon is properly selected. Also, note that the method has optional key-word arguments which determine whether the dialog box is displayed, and which determine which output data objects will be created when the dialog box is not displayed. These optional argument permit keyboard entry of the statement

```
(send current-model :create-data :dialog t)
```

which presents the user with the dialog box. This is the statement which is generated by the menu system. On the other hand, these arguments permit the statement:

```
(send current-model :create-data
  :scores t :coefs nil)
```
which creates one output data object of scores without the user having to intervene by responding to a dialog box. Thus, this statement can appear in a script file. A create-data method for another model object must have a similar construction if it is to work in all of these ways.

This method calls three other methods to create the data objects. One of these methods (the :create-input-data-object method) is already defined by mv-model-object-proto, and is a method which creates a copy of the input data, taking into consideration which variables are selected and active. The other two methods are specific to the principal components model, but it is instructive to show one of these here (they are both nearly identical). The :pca-scores-data-object method is:

```
(defun pca-model-object-proto
  :pca-scores-data-object (creator)
  (data (concatenate 'string "Scores-"  
    (send self :name))
  :created creator
  :title (concatenate 'string "PCA Scores for "  
    (send self :title))
  :data (combine (send self :scores))
  :variables
    (mapcar #'(lambda (x) (format nil "PC~a" x))  
      (iseq (min (send self :nvar) (send self :nobs)))))
  :labels (send self :labels)
  :types (repeat "Numeric"  
    (min (send self :nvar) (send self :nobs)))
)
```

Note that the method consists entirely of a data function to create a new data object (this is the same data function used in data files). Various functions are used to create the value for the arguments of the function. The name is created by concatenating a meaningful prefix (in this case “scores”) with the model object’s name. The title is created by concatenating a similar prefix with the model object’s title. The data consist of a list of scores, the combine function being used to convert the scores matrix to a list. The variable names are a combination of the string “PC” and a sequence number. The labels and types are self-explanatory. If this method is copied for new model objects, then these new model objects will produce data objects that conform with other ViSta data objects.
Report: The report method for the principal components prototype is:

```lisp
(defun pca-model-object-proto :report  
  &optional (stream t)  
  (let* ((w nil)  
         (labels (send self :labels))  
         (vars   (send self :variables))  
         (scores (send self :scores))  
         (coefs  (transpose (send self :coefs)))  
         (eigenvalues (send self :eigenvalues))  
         (proportions (/ eigenvalues (sum eigenvalues)))  
         (fitmat (transpose  
                   (matrix (list 3 (min (send self :nobs)  
                                      (send self :nvar)))  
                             (combine eigenvalues proportions  
                              (cumsum proportions)))))))  
    (lc-names  
               (mapcar #'(lambda (x) (format nil "PC~a" x))  
                    (iseq (send current-model :nvar))))  
    (+macintosh(def w (send display-window-proto :new :title  
                                  "ViSta Report")))  
    (-macintosh(display-string (format nil "ViSta Report  
                                 ****************~2%"))  
                 (display-string  
                                 (format nil "~2%Principal Components Analysis~2%") w)  
                 (display-string  
                                 (format nil "~2%Model: ~a~2%" (send self :name ) ) w)  
                 (display-string  
                                 (format nil "~2%Variable Names: ~a~2%" vars) w)  
                 (display-string  
                                 (format nil "~2%Fit Indices for each Component:~%Eigenvalue (amount of total data variance fit by each component)~%Proportion (of total data variance fit by each component)~%Cumulative Proportion (of total data variance fit by the components)~2% Eigenvalue Proportion Cum Propor Component~%") w)  
                 (print-matrix-to-window fitmat w :labels lc-names)  
                 (display-string  
                                 (format nil "~2%Coefficients (EigenVectors):~%") w)  
                 (print-matrix-to-window  
                                 (fuzz coefs) w :labels lc-names)  
                 (display-string  
                                 (format nil "~2%Component Scores:~%") w)  
                 (print-matrix-to-window  
                                 (fuzz scores) w :labels labels) w))
```
This function is used by the menu-system, as well as by the user from either the keyboard or from script files, by the statement: (send current-model :report). Note that the method contains the statement (if (not (eq current-object self)) (setcm self)) which is used for the same reason that it was used in the create-data-objects method. The let* statement is used to locally define those values which are to be reported. Then, the #+macintosh(def w ...) statement is used to define, for only the Macintosh, the window w in which the report appears. This is defined only for the Macintosh because other systems, including MS-Windows and X-Windows do not have display windows. For those systems a heading is printed on the standard output by the statement that begins #+macintosh. In constructing other reporting methods, you should use these exactly these same statements so that the report’s appearance is consistent across methods. Next, a series of display-string and print-matrix-to-window functions are used to report the desired information. These functions should be used in place of standard printing functions since they write to the Macintosh display window or the MS-Windows or X-Windows standard output.

**Visualize:** We do not present the visualize method here because it is too long. The important aspect of it for those constructing the method for other models is that it has no arguments, and is used by the menu-system and from the keyboard by the statement (send current-model :visualize).

**Step 4: Define the Model’s Constructor function:** You must create a constructor function to construct an instance of the model object prototype. This is the function that is actually used to analyze data. It is used by the menu system or by the data analyst via the keyboard or from script files to construct an instance of the model object. The constructor function for principal components is:

```lisp
(defun principal-components
  (&key
    (data        current-data)
    (title       "Principal Components")
    (name        (concatenate 'string "PCA-" (send current-data :name)))
    (dialog      nil)
    (covariances nil))
  (if (not (eq current-object data)) (setcd data))
  (send pca-model-object-proto :new (not covariances) 6 data title name dialog))
```

This function uses keyword arguments. The defaults are such that if the analyst types the statement (principal-components) the analysis will be performed on correlations computed from the active numeric variables in the current data object. The dialog box will not be presented. The methods menu generates the statement (principal-components :dialog t), so that the dialog box is presented to see if the user wishes to perform the analysis on correlations or covariances.

ViSta: The Visual Statistics System
The constructor functions for other models should be similar to this one. Since all other constructor functions in ViSta use keyword arguments, it is recommended that new models also have keyword arguments. The first four keywords must be used with all models as they appear in the example since they are processed by the underlying data object prototype. Of course, the values of the title and name keywords should be replaced with appropriate strings to identify the model. The dialog keyword should also be used as is, with the default value nil. The covariances keyword is specific to the components model. For other models it would be replaced with other keywords specific to the other models.

The send statement at the end of the constructor function should appear as it does here for other models, except for the presence of (not covariances), which must be replaced by arguments corresponding to the keywords which are specific to the model. The value 6 corresponds to the number of the tool icon. Consult the author for how this is changed for other models.

**Step 5: Define the :isnew method:** The next step is to define the model’s :isnew method, the method which is invoked by the :new message in the model’s constructor function. For principal components this method is:

```lisp
(defmeth pca-model-object-proto :isnew (corr &rest args)
  (cond
    ((> (send current-data :active-nvar '(numeric))
        (send current-data :nobs))
      (error-message "Note: Cannot analyze data with fewer observations than variables."
        (send *toolbox* :reset-button 6))
    (t
      (send self :model-abbrev "PCA")
      (send self :corr corr)
      (apply #'call-next-method args))))
```

First, the :isnew method creates a new instance of the model. This is done by the Lisp-Stat object system; there are no statements in the :isnew method that correspond to this action. The first thing that explicitly takes places in the :isnew method is that the data are checked to see if there are more active variables than observations. If so, an error message is issued, the appropriate method button (number 6) is reset, and the analysis is terminated. If there are at least as many observations as variables, the model abbreviation is set to PCA, and the value (t or nil) of the corr argument is stored in the corr slot. Then call-next-method is applied to each of the remaining arguments, calling the isnew method for mv-model-object-proto.

This general flow should be followed in any other model’s :isnew method: First, check on the validity of argument values. If they are not valid, report an error and
reset the button. If they are valid, store the model abbreviation and argument values in their slots. Then, apply call-next-method to the remaining arguments.

**Step 6: Define the Method Menu Item and Button:** Code to create a new item for the method menu must be added to the menus.lsp file. See how this is done for existing menu items in that file to determine how a new item is created. For principal components the statement is:

```
(setf prin-model-menu-item
    (send menu-item-proto :new "Principal Components"
       :action #'(lambda ()
                    (principal-components :dialog t))
       :enabled nil))
```

You must then add the name of the model item (prin-model-menu-item, in this case) to the list of menu item names associated with the (send *tools-menu* :append items) statement that appears in the middle of the menus.lsp file.

In order to have the menu item enable and disable according to the type of the current data object, you must modify the setcd function in the dataobj.lsp file. For principal components, which is appropriate to multivariate data, but not to dissimilarity and table data, the statement (send prin-model-menu-item :enabled t) was added to the portion of setcd’s cond function that is true for multivariate data, and (send prin-model-menu-item :enabled nil) was added to the portions that are nil for multivariate data.

In order to have the method’s tool-bar button enable and disable according to the type of the current data object, you must further modify the setcd function in the dataobj.lsp file. For principal components (whose button is number 6) the statement (send (select tools 6) :icon-state "normal") was added to the portion of setcd’s cond function that is true for multivariate data, and (send (select tools 6) :icon-state "gray") was added to the portions that are nil for multivariate data. A method for adding new method buttons has not yet been developed, thus, if your new method does not correspond to an unused button, it cannot yet be represented by a button.
Step 7: On Demand Code Loading Feature. The code for a model-object is not loaded until the first time that an instance of the object prototype needs to be constructed. Usually, this is when the object’s method menu or method button is used. This delayed, on demand, code loading feature is created by the loading functions that appear at the end of the modelobj.lsp file. The loading function for principal components is:

```
(defun principal-components
  (&key
    (data        current-data)
    (title       "Principal Components")
    (name        (concatenate 'string "PCA-
                              (send current-data :name)))
    (dialog      nil)
    (covariances nil))
  (load (concatenate 'string *vista-dir-name*
                        "pcamob")
        (principal-components
         :data data
         :title title
         :name name
         :dialog dialog
         :covariances covariances))
```

Notice that the loading function has exactly the same name, argument list and default values as the constructor function. This must be true for the loading function for every model. The body of the loading function must always consist of two statements. The first uses the load function to load the desired model object file. This file should be located with all other ViSta files, and is accessed by concatenating *vista-dir-name* (whose value is the directory path) with the name of the file (pcamob, in this case). The second statement is the constructor function which constructs the model object. All options must be explicitly specified, and they must be given values obtained from the argument list of the loading function.
Creating New Model Objects