ViSta

Developing Statistical Objects

Forrest W. Young

October, 1996
Research Memorandum Number 96-1

THE L.L. THURSTONE
PSYCHOMETRIC LABORATORY
UNIVERSITY OF NORTH CAROLINA

CHAPEL HILL, N.C.
U.S.A. 27599-3270

ViSta

Developing Statistical Objects

Forrest W. Young

The L.L. Thurstone
Psychometric Laboratory
University of North Carolina
Chapel Hill NC 27599-3270

ViSta: Developing Statistical Objects

The correct reference for this document is:
Young, F.W. (1996) ViSta: Developing Statistical Objects. Research Memorandum 96-1. L.L.Thursone Psychometric Laboratory, University of North Carolina. Chapel Hill, NC.

ViSta is available free of charge from:
www: http://forrest.psych.unc.edu/research/vista.html
ftp: ftp.psych.unc.edu/pub/forrest/vista

For more information contact the author at:

UNC Psychometrics
CB 3270 Davie Hall
Chapel Hill NC 27599-3270

Phone: 919-962-5038
e-mail: forrest@unc.edu
www: http://forrest.psych.unc.edu/

For ViSta version ß4.25 (October, 1996)
Printed October 28, 1996

ViSta: Developing Statistical Objects

This monograph is designed to guide those who are programming new data or model objects (statistical objects) for ViSta, the Visual Statistics system (Young, 1996).

The monograph overviews the inheritance structure of ViSta’s data and model object systems; discusses ViSta's global variables; presents details on the methods, functions and messages that can be used with statistical objects; reviews the steps taken during the construction of an instance of a model object; and presents a detailed example of how one of ViSta’s model objects was developed.

This monograph assumes you are familiar with Lisp-Stat (Tierney, 1990).

1 ViSta's Statistical Object System

ViSta's statistical object system is based on Lisp-Stat's object-oriented programming system. The statistical object system consists of prototype objects which have methods (pieces of LispStat object-oriented code) and information designed with statistical data analysis in mind. The prototype objects permit the instantiation of the statistical objects which are represented on the WorkMap by icons. The objects use the methods to send and receive messages that access and manipulate the information. We discuss the statistical object system in this chapter.

ViSta's prototype statistical objects fall into two major categories: prototype data objects and prototype model objects. There are three different kinds of data prototypes, each specialized for one kind of data — multivariate, matrix and table data. There are several kinds of model prototypes, each specialized to one way of modeling data. Examples include the analysis of variance prototype, the two regression prototypes, the univariate analysis prototype, etc.

The statistical prototypes are related to each other hierarchically to take advantage of the Lisp-Stat object system's inheritance feature. The hierarchical structure is shown in Figure 1. Note that the model prototypes inherit from mv-model-object-proto, an abstract model prototype which contains methods and information common to all model prototypes. It in turn inherits from mv-data-object-proto, which contains methods and information used by all statistical objects.

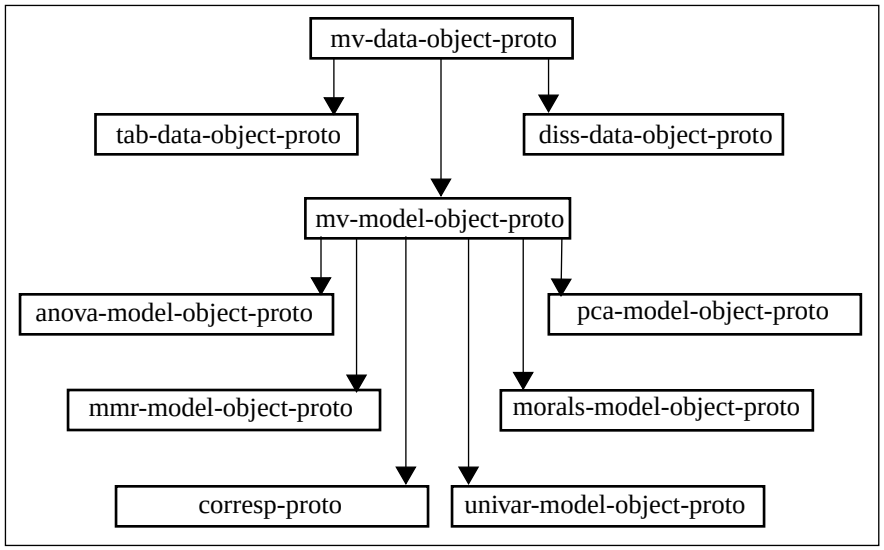

Figure 1: Statistical Object Prototype Hierarchy

2 Global Variables

ViSta defines a number of global variables — variables whose values are always available for use. We present them in this section

One of the statistical objects is always the "current object". The current object is indicated in the workmap by the high-lighted icon. It is also always pointed to by the global variable `*current-object*`, to which messages can be sent. The current object can be changed by clicking on a new icon on the workmap, by choosing a new object from the data or model menus, or by using the `setcd` or `setcm` functions described in sections 3.2 and 4.1. The global variables `*current-data*` and `*current-model*` are also discussed in those sections.

There are several global variables that point to special directories in your computer's file system. These are `*startup-dir*`, which points to the directory containing the LispStat load module; `*vista-dir-name*` whose value is the directory containing the basic vista code and sub-directories. The variables `*data-dir-name*`, `*help-dir-name*`, `*guide-dir-name*`, and `*example-dir-name*` specify ViSta's data, help, guidance and example directories.

In addition, there are global variables whose values point to various windows and menus. These include `*workmap*`, `*guidemap*`, `*expertmap*`, `*obs-window*` (also referred to by `*mat-window*`), `*var-window*`, `*edit-menu*`, `*file-menu*`, `*command-menu*`, `*data-menu*`, `*trans-menu*`, `*analyze-menu*` and `*model-menu*`. In addition, `*vista*` points to the vista system object, and `*copyright*` points to the copyright message window. Many of these are nil until used.

3 Data Objects

Data objects are used to define data used by ViSta. In this section we discuss the `data` function for defining data objects, and the keywords which can be used with the function. We then discuss data object types, the concept of the current data object, and the messages which can be sent to an existing data object. Note that the `(load-data)` function may be used to load data into ViSta from a ViSta datafile, the `(open-data)` function loads data into ViSta from a ViSta datafile and then displays it as a datasheet, and the `(import-data)` function imports data contained in a text file. Each of these functions takes an optional string argument to specify the name of the file containing the data. Thus, it is possible to load data from the data directory, by using the global variable `*data-dir-name*`, with the statement

```
(load-data (strcat *data-dir-name* "cars.lsp"))
```

These menu items are all discussed in Chapter 3.

3.1 The Data Function

The several ways of creating data that we discussed in section 3.3 of Chapter 3 of Young (1996) all create data by using the `data` function. This is an object-constructor function which creates an instance of a data object. It can create any of the three types of data objects – multivariate, matrix or table data objects. The specifics of the `data` function for each type of data object are explained in the each of the next three subsections.

```
(data "HealthClub"
:title
"Data from a Health Club"
:variables
'("Weight" "Waist" "Situps")
:data '(
191 36 162
189 37 110
193 38 101
162 35 105
189 35 155
182 36 101
211 38 101
167 34 125
176 31 200
154 33 251
169 34 120
166 33 210
154 34 215
193 36  70
202 37 210
176 37  60
157 32 230
156 33 225
138 33 110))
```

Figure 2: Data function for multivariate data

3.1.1 Multivariate Data

An example of the `data` function being used to create an instance of a multivariate data object is shown in Figure 2. The `data` function has one initial required argument and two required keyword arguments. The initial required argument is a string that is used to name the data. In the Figure, this is "`HealthClub`". Any characters may be used in this string, including spaces. In addition to the name, you must also use the `:variables` and `:data` keywords, following each with a list of elements. The `:variables` keyword specifies the names of the variables and (indirectly) the number of variables. It is followed by a list of character strings which are the variable names. The `:data` keyword specifies the list of data values. If *n* is the number of variables, the first *n* elements of the list are the values for the first observation (row) of the data, the next *n* elements are the values for the second observation, etc. The total number of elements of the data list must be an exact integer multiple of the number of variables.

There are several optional keywords which may be used with the `data` function, one of which, the `:title` keyword, is shown in the example. This keyword is followed by a character string to specify the data-object's title, which is used in various windows. When not specified, the data title is `Untitled Data Object`. The `:types` keyword, which is followed by a list of character strings, one for each variable, specifies the type of each variable. The type may be "`category`", "`ordinal`", or "`numeric`". If this keyword is not used, all variables are assumed to be numeric. Finally, the `:labels` keyword, which is followed by a list of character strings, specifies the label for each observation (row of the data). If not used, the observation labels are `Obs0`, `Obs1`, etc.

3.1.2 Matrix Data

An example of a `data` function for creating an instance of a matrix data object is shown in Figure 3. A matrix data object is a data object whose data consist of one or more matrices of data. ViSta matrices are not as general as those in Lispstat: They must be square, and the row and columns must refer to the same set of things, which are named by the `:variables` keyword.

```
(data        "FlyMiles"
:title       "Flying Mileages between 10 Cities"
:variables '("atlanta" "chicago" "denver" "houston"
             "los angeles" "miami" "new york"
             "san francisco" "seattle" "wash.d.c.")
:labels    '("atlanta" "chicago" "denver" "houston"
             "los angeles" "miami" "new york"
             "san francisco" "seattle" "wash.d.c.")
:matrices  '("Mileages")
:data      '(
   0  587 1212  701 1936  604  748 2139 2182  543
 587    0  920  940 1745 1188  713 1858 1737  597
1212  920    0  879  831 1726 1631  949 1021 1494
 701  940  879    0 1374  968 1420 1645 1831 1220
1936 1745  831 1374    0 2339 2451  347  959 2300
 604 1188 1726  968 2339    0 1092 2594 2734  923
 748  713 1631 1420 2451 1092    0 2571 2408  205
2139 1858  949 1645  347 2594 2571    0  678 2442
2182 1737 1021 1831  959 2734 2408  678    0 2329
 543  597 1494 1220 2300  923  205 2442 2329    0
))
```

Figure 3: Data function to define matrix data.

Matrix data objects are defined in exactly the same way as multivariate data objects, except that the `:matrices` keyword must be used. This keyword, which is followed by a list of character strings, specifies that the data are matrix data, and specifies the names (and, indirectly, the number) of the matrices. The :`matrices` keyword is required for matrix data, and must not be used for other types of data. The `:shapes` keyword, which is followed by a list of character strings, one for each matrix, specifies the shape of each matrix. The shapes may be `"symmetric"`, or `"asymmetric"`. If shapes are not specified, then all matrices are assumed to be symmetric. In the example shown in Figure 3, both the `:shapes` and the `:types` keywords are not needed, because they both specify characteristics of the data which are assumed by default. If the data consists of several matrices, then the `:data` list consists of all the values for the first matrix, followed by all the values for the second matrix, etc.

3.1.3 Table Data

Instances of table data objects are defined in the same way as previous data objects except that only one variable can be specified by the `:variables` keyword (i.e., the data must be univariate), the `:data` keyword must be followed by a list of lists (rather than a list of values), and the `:classes` and `:ways` keywords must be used.

```
(data "Eggs2"
      :variables '("FatContent")
      :ways '("Lab" "Sample")
      :classes '(("A Lab" "B Lab" "C Lab"
                          "D Lab" "E Lab" "F Lab")
                 ("1st Sample" "2nd Sample"))
      :data '(

;        Sample 1                  Sample 2
   (0.62 0.55 0.80 0.68)     (0.34 0.24 0.76 0.65);  Lab A
   (0.30 0.40 0.39 0.40)     (0.33 0.43 0.29 0.18);  Lab B
   (0.46 0.38 0.37 0.42)     (0.27 0.37 0.45 0.54);  Lab C
   (0.18 0.47 0.40 0.37)     (0.53 0.32 0.31 0.43);  Lab D
   (0.35 0.39 0.42 0.36)     (0.37 0.33 0.20 0.41);  Lab E
   (0.37 0.43 0.18 0.20)     (0.28 0.36 0.26 0.06);  Lab F
))
```

Figure 4: Data function to define two-way table data.

An example is shown in Figure 4. The `:ways` keyword is followed by a list of character strings that are used to name the ways of the data. In the example the ways are "`Lab`" and "`Sample`". The arguments of the `:classes` keyword specify the names and the number of levels of each way of the table. For one-way data the `:classes` keyword is followed by a list of character strings. For multi-way data it is followed by a list of lists of character strings. The number of lists must correspond to the number of ways. In the two-way example shown in Figure 4, the `:classes` keyword is followed by a list of two lists. Since the first list has six elements and the second list has two, these data form a 6x2 table. Finally, the `:data` keyword must be followed by a list of lists of values. Each sub-list is a cell of the design. The list of lists allows varying numbers of data-elements per cell. In the example, there are 4 elements in each list. Thus, these data are balanced (same number of observations in each cell) 6x2 two-way data that are replicated 4 times.

A second example appears in Figure 5. In this example the `:classes` list has only one list of four classes, hence the data are one way, and the way has four levels. Note that the `:ways` keyword specifies one way. The data consist of four sub-lists of different lengths. Since each sublist specifies all of the multiple observations for each cell of the table, these data are unbalanced.

```
(data        "Singers"
:title       "Singer's Heights"
:variables '("Height")
:types     '("Numeric")
:ways      '("Part")
:classes   '(("Sopranos" "Altos" "Tenors" "Basses"))
:data      '(
(64 62 66 65 60 61 65 66 65 63 67 65 62 65 68 65 63 65
 62 65 66 62 65 63 65 66 65 62 65 66 65 61 65 66 65 62)
(65 62 68 67 67 63 67 66 63 72 62 61 66 64 60 61 66 66
 66 62 70 65 64 63 65 69 61 66 65 61 63 64 67 66 68)
(69 72 71 66 76 74 71 66 68 67 70 65 72 70 68 73 66 68
 67 64)
(72 70 72 69 73 71 72 68 68 71 66 68 71 73 73 70 68 70
 75 68 71 70 74 70 75 75 69 72 71 70 71 68 70 75 72 66
 72 70 69)))
```

Figure 5: Data function for one-way unbalanced table data

3.2 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 be indicated in the workmap as the high-lighted data icon, if one is high-lighted. The current data's object identification can always be found in the global variable `*current-data*`, to which messages can always be sent. The current data can be changed by clicking on a new data icon, or by choosing a new item of the data menu.

The `data` function 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 `setcd` (set current data) function. For example, if there is a data object named `cars`, you make it the current data with `(setcd cars)`.

3.3 Data Object Messages and Functions

Messages can be sent to data objects. The messages can be sent to `*current-data*` or to the name of a specific 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 `cars`, you could type `(send cars :data)` to see a listing of its data values, whether or not it is the current data.

Slot Information Messages: There is a group of messages[1] which can be used to obtain information about the data object. For example `(send cars :title)` causes the data-object to tell you its title. In addition to the `:title` message, messages in this group are `:nobs`, which returns the number of observations in the data object; `:nvar`, which returns the number of variables in the object; `:variables`, which returns the names of the variables; `:labels`, which returns the labels of the observations; `:types`, which returns the type of each variable; `:data`, which returns the data as a list; `:data-matrix`, which returns the data as a matrix; and `:name` which returns the name of the data object. The messages `:obs-states` and `:var-states` return the state of each observation or variable in the observation window (the states can be normal, selected or invisible). For matrix data, the message `:matrices` can be used to obtain the names of the matrices, and you can use the messages `:nmat, :mat-states` and `:shapes` (to see whether the matrices are symmetric or asymmetric). For table data, you can use the messages `:nways`, `:nclasses`, `:ncells`, `:cellfreqs`, `:classes`, `:source-names`, `:level-names`; and `:indicator-matrices`.

1. These messages are slot-accessor messages, and therefore can also be used to change the information in data object slots. This should not be done since ViSta will not work correctly if the information is changed.

Data Menu Item Messages and Functions: There is another group of messages which perform the same actions as those performed by items of the data menu. The name of each message corresponds to the name of a menu item. For example, the data menu's **Save Data** menu item corresponds to `(send *current-data* :save-data)`. The messages include `:save-data`, which saves data into a ViSta datafile specified in a dialog-box; and `:create-data`, which creates a new data object from the active portion of the current data. These two messages can be followed by a string argument to name the file or data object. If the string is not specified, a dialog box is presented. The message `:browse-data` shows the datasheet; `:edit-data` shows the datasheet and enables editing of the data; `:report-data` creates a report (listing) of the data; `:list-variables` shows a window with a list of the variables in the current data (you can also use `:list-vars` or `:list-var`); and `:list-observations` (or `:list-obs`) shows a window with a list of the observations in the current data. For matrix data you may use `:list-matrices` (or `:list-mats` or `:list-mat`) to show a window with a list of the matrices in the data. For table data you may use `:list-cells`.

Each message in the previous paragraph also has a short form, called a generic function. For example, `(save-data)` is short for `(send *current-data* :save-data)`. The long-form message can be used to send messages to a specifically named data object that is not the current data. The short-form generic function always sends the message to the current data.

There are four menu items which have a short-form that effects the `*current-data*`, but do not have a long-form message that can be sent to data objects that are not current. These are `(visualize-data)`, which visualizes the active data, `(merge-variables)`, which merges variables using the active data in the current data and the active data in the previously current data; `(merge-observations)`, which merges observations using the active data in the current data and the previously current data, and `(merge-matrices)`, which merges matrices using the active data in the current and previously current data (both have to be matrix data). The merge functions 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.

You can also use the `:summarize-data` message or the `(summarize-data)` function to see a summary report (listing of summary statistics) of the data. The message and function each have five keyword arguments, each of which must be followed with `t` or `nil` (the default). The arguments are `:moments`, `:quartiles`, `:ranges`, `:correlations`, `:covariances`, and `:dialog`. The last argument determines if a dialog box is presented to obtain the desired types of statistics. The others determine which statistics are reported if no dialog is shown.

Active Data Messages: There are numerous messages which deal with the “active” data: i.e., the subset of the current data which is specified by selections in the windows which list observations, variables and matrices. The active data consists of the subset of data elements whose variable (or matrix) names and observation labels which appear in the "Vars" (or “Mats”) and “Obs” windows, or, if any names or labels are selected in a window, those which are selected as well as visible. The concept of active data does not apply to table data, which may not be subsetted.

The messages `:select-variables`, and `:select-observations`, and the short-form functions `(select-variables)` and `(select-observations)` can be used to select the variables or observations which are to be active. Similarly, with matrix data you may select matrices with the message `:select-matrices` or the function `(select-matrices)`. Each message or function must be followed by a list of strings specifying the names of the variables or matrices or the labels of the observations that are to be activated. For example, you can select all of the variables with

```
(select-variables (send *current-data* :variables)).
```

There are several messages that provide information about the selected subset of data. Many of these messages take an argument that is a list of symbols (not strings) that specifies the variable “type”. The type symbols are `labels`, `category`, `ordinal`, `numeric` or `all` (in upper, lower or mixed-case). The messages using these symbols return information about the data elements whose active variables have types which match the specified types. For example, you could type

```
(send cars :active-data '(ordinal category))
```

to see a list of the data elements whose variables are active and are either ordinal or categorical. The messages that use type-symbols are `:active-data` (to get a list of the active portion of the data), `:active-data-matrix` (to get a matrix of the active portion of the data), `:active-nvar` (to find out how many active variables there are), `:active-variables` (to obtain a list of the names of the active variables), and `:active-types` (to obtain a list of the types of the active variables). In addition, the messages `:active-matrices`, `:active-shapes`, and `:active-nmat` take an argument that is a list of the shape symbols `all`, `symmetric`, and `asymmetric`. Finally, there are two active data messages `:active-labels` and `:active-nobs` do not have an argument. These messages can be used to get a list of the labels of the active observations, or the number of active observations. Note that there are no short-form functions for any of the active data messages mentioned in this paragraph.

Miscellaneous Messages: The message `:variable` reports values of the variable whose name is the argument of the message. The messages `:means`, `:standard-deviations`, `:variances`, `:skewnesses`, `:kurtoses`, `:minimums`, `:medians`, `:maximums`, `:ranges:`, `:mid-ranges`, `:interquartile-ranges`, and `covariance-matrix` report simple statistics of the active data. There is no short-form function for these messages.

4 Model Objects

Model objects are used to define models in ViSta. There are several functions for constructing instances of model objects, one function for each of the model object prototypes given in Figure 1. As shown in the figure, each prototype inherits methods and information from `mv-model-object-proto`, the general model object prototype that contains methods and information useful to all model objects. This prototype in turn inherits all of the methods and information in the `mv-data-object-proto` that were discussed in the previous section. Thus, the messages discussed in section 3 can be sent to model objects as well as to data objects.

4.1 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 is shown in the workmap when a model icon is the high-lighted 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-cars`, you can type `(setcm pca-cars)` to change the current model to `pca-cars`. You can also get information about a model by sending messages to any model. For example, you could type `(send pca-cars :title)`.

4.2 ViSta Model Object Messages and Methods

As noted above, all model objects have many methods inherited from `mv-data-object-proto` and from `mv-model-object-proto`. However, some methods that are needed by all model objects are not inherited from the ancestor objects. These must be uniquely defined for each individual model object so that their actions are appropriate to the particular model. These methods are called ViSta model object methods. These methods are:

1. `:options` — shows a dialog box to obtain values for the options of the analysis method. It places options values in slots that are unique to the model object.
2. `:analysis` — performs the analysis. It reads the information in the options slots and places results in analysis slots that are unique to the model object.
3. `:save-model-template` — used by the **Save Model** menu item.
4. `:create-data` — used by the **Create Data** menu item.
5. `:report-model` — used by the **Report Model** menu item.
6. `:visualize-model` — used by the **Visualize Model** menu item.

If you are planning on writing your own ViSta model object, you will have to write these methods yourself, as well as some additional methods and functions. We show an example of how this is done in section 5.

4.3 Steps Taken During Object Construction

When the user requests a data analysis (by using the **Analysis** menu, a toolbar button, a script, or by typing in the listener) ViSta takes a series of actions. These actions begin with the invocation of the model's constructor function.

The model's constructor function first checks on the validity of its argument values. If they are valid it then issues the `:new` message to the prototype model object. This message must be followed by the following set of arguments: The first several arguments are the parameters specific to the particular model (such as `covariances` for the principal components model). Then, there must be five arguments: 1) an integer which specifies the number of the model's method-button; 2) the object identification information for the data object being analyzed; 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 prototype's `:isnew` method. The first few arguments of this method correspond to the arguments that were used in the constructor function's `:new` message which are unique to the model. Then (`&rest args` can be used for the last several arguments. This method creates a new model instance. Then, the values of the arguments that are unique to the model must be saved in the instance's slots. The method must then apply `call-next-method` to `args`. This calls `mv-model-object-proto's :isnew` method with the arguments that it needs. Then `mv-model-object-proto's :isnew` method takes the following actions:

1. The `mv-data-object-proto`'s `:isnew` method is called via `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 analysis icon is added to the workmap when the `:copy-tool-icon` message is sent to the `*toolbox*`.
3. The `mv-data-object-proto`'s dialog slot is set to `t` or `nil`, depending on the value of `dialog` that was used in the constructor function. This determines whether a dialog box will be presented.
4. The model's `:options` method is used. If `dialog` is `nil`, it does nothing. If it is `true`, the `:options` method shows 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 model object's slots, ignoring the values that are already there. The `:options` method returns `nil` when it is not used right, or when it is can-

celed, so that the analysis will be halted. Otherwise, the value returned by the `:options` method is ignored.

5. Now `mv-model-object-proto` determines whether to proceed with the analysis. It proceeds when the `:options` method returned a non-nil value.
6. The `mv-data-object-proto`'s `:data-object` method saves the object identification information about the data that are being analyzed in a slot that exists in `mv-model-object-proto`.
7. The `:analysis` method is invoked by `mv-model-object-proto`. The `:analysis` method uses whatever is in the model object's slots to carry out the analysis. Note that the `:analysis` method doesn't directly use the dialog results, rather, it uses information that has been put into slots by the dialog. More generally, none of the ViSta system methods use the results of any other method directly. All communication between methods is done via slots.
8. The `mv-data-object-proto`'s `:new-object` method updates the system to recognize the new model object. It adds a model icon to the workmap and a menu item to the model menu. The `*current-model*` and `*current-object*` global variables are updated to point to the new model. The variable and observation windows are cleared. A variable is created with the name of the new model and value of the model's object identification.

5 An Example: The Principal Components Model Object

In this section we show the steps that were taken to write ViSta's Principal Components model object prototype. 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. Note that the steps given here appear in the order that they were written, not in the order that they are used by the system.

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` 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. 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 of the principal component scores and coefficients, and the `:eigenvalues` slot will contain a vector of 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 4.2, each model prototype must have 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-model` and `visualize-model` (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 prototype is given below.

```
(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) (setf 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 to nil by default, 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. For an example of a more complex dialog, look at the code for the multivariate multiple regression options dialog.

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))
         (nobs (select (array-dimensions data) 0))
         (prepped-data
           (if (send self :corr)
             (/ (normalize (center data) 1)
                (sqrt (1- nobs)))
             (/ (center data) (sqrt (1- nobs)))))
         (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)
    t))
```

The details of this method are not important for this example, other than to note that all of the analysis results (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. The method communicates its results indirectly through slots, not directly by returning them. The method returns t.

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. The `mv-model-object-proto`'s `:save-model` method takes information placed into the template (as explained below) and places it in a file. When the file is loaded back into ViSta by `(load-model)`, a principal components analysis is performed on the data that were also saved in the file.

For principal components, the method is:

```
(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 places the principal-components function, with the functions following commas being replaced with their results, in a file. When the file is loaded back into ViSta by (load-model), a principal components analysis is performed on the data that were also saved in the file.

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 shown below.

There are several things to point out about this method. First, 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 arguments permit the menu system to generate:
(send *current-model* :create-data :dialog t).
This causes the dialog box to be shown. On the other hand, these arguments permit a script to contain 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. For the system to work in these ways, any create-data method for another model object must have a similar construction.

Also, note the statement, near the beginning of the function:
`(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 is properly selected. Finally, `(not (not desires))` causes the method to return `nil` when the dialog is canceled, and `t` when it is not canceled.

```
(defmeth 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)))
    (not (not desires))))
```

This method calls three other methods which actually create the data objects. One of these (`:create-input-data-object`) is already defined by `mv-model-object-proto`. It 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. One of these is shown here (they are nearly identical):

```
(defmeth pca-model-object-proto)
         :pca-scores-data-object (creator)
  (data (strcat "Scores-" (send self :name))
    :created creator
    :title (strcat "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 `pca-scores-data-object` method consists entirely of a data function to create a new data object (this function was discussed earlier in this chapter). The arguments of the function were described earlier in the chapter. except for `:creator`. This argument specifies which workmap icon object represents the object which is creating the data object, so it's argument must be the object identification of the appropriate icon. This information is used to construct the workmap.

Various functions are used to create the values 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.

Visualize-Model: We do not present the `visualize-model` 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-model).
```

Report-model: The `report-model` method for the principal components prototype is shown below. 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-model).
```

The method uses three special functions which ensure that the report works correctly under the various operating systems, and that it's appearance is consistent between models. These functions are `report-header`, `display-string`, and `print-matrix-to-window`. In constructing other reporting methods, you should use these functions in place of standard printing functions since they write to the Macintosh text window or to the MS-Windows or X-Windows listener in a consistent fashion. The statement

```
(report-header (send self :title))
```

determines how and where the report should be printed, which varies from one operating system to another. It displays the title and returns a value which identifies the place that the information should be written (a text window or the listener). In the example, the variable w gets this value. The `display-string` and `print-matrix-to-window` functions are used to report the desired information in the desired way. The `display-string` function takes a string as its first argument, and `w`, the window identification, as its second argument. The `print-matrix-to-window` function has two required arguments: The first is the matrix which is to be printed, the second is `w,` the place it is to be printed. This function also has

two optional arguments. One is `:labels`, which must be followed by a list of strings. These are used to label the rows of the matrix. The other is `:decimals`, which must be followed by an integer. The integer determines the number of decimals that are printed following the decimal point.

Finally, note that the method contains the statement

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

to ensure that the proper object is reported.

```
(defmeth pca-model-object-proto :report
                        (&key (dialog nil))
  (if (not (eq *current-object* self)) (setcm self))
  (let* ((w (report-header (send self :title)))
         (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)))))
    (display-string
     (format nil "Principal Components Analysis of") w)
    (if (send self :corr)
      (display-string (format nil " Correlations~2%") w)
      (display-string (format nil " Covariances~2%") w))
    (display-string
      (format nil "Model: ~a~2%" (send self :name )) w)
    (display-string
      (format nil "Variable Names: ~a~2%" vars) w)
    (display-string
      (format nil "Fit Indices for each Component:~
                ~2% Eigenvalue  Proportion  ~
                    Cum Propor Component~%") w)
    (print-matrix-to-window fitmat w
                :labels lc-names :decimals 5)
    (display-string
      (format nil "~%Coefficients (EigenVectors):~%") w)
    (print-matrix-to-window coefs w
                :labels lc-names :decimals 3)
    (display-string
      (format nil "~%Component Scores:~%") w)
    (print-matrix-to-window scores w
                :labels labels :decimals 3)
    w))
```

Step 4: 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:

```
(defmeth pca-model-object-proto :isnew (corr &rest args)
  (cond
    ((> (send current-data :active-nvar '(numeric))
        (send current-data :active-nobs))
     (error-message
       (format nil "Note: Cannot analyze data ~
                with fewer active observations (~d) ~
                than active numeric variables (~d)."
           (send current-data :active-nobs)
           (send current-data :active-nvar '(numeric))))
     (send *toolbox* :reset-button 6))
    ((< (send current-data :active-nvar '(numeric)) 2)
     (error-message
       (format nil "Cannot analyze data ~
                that has less than two active ~
                numeric 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 active observations or less than two active numeric variables. If so, an error message is issued, the appropriate method button (number 6) is reset, and the analysis does not take place. Note that the `(error-message)` function displays the message in a dialog box. 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, use `(error-message)` to report an error. Then 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 5: 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 the data. It is used by the menu system, by the toolbar, by the data analyst via the keyboard and by script files. The constructor function for principal components is:

```
(defun principal-components
  (&key
     (data       *current-data*)
     (title      "Principal Components")
     (name        nil)
     (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 analysis 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.

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 `mv-model-object-proto`. Of course, the default value of the `title` should be replaced with appropriate strings to identify the model. A default value for the name keyword is automatically constructed by concatenating the three-letter model abbreviation with the name of the data. 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.[2]

2. This feature will be modified in the near future. In the meantime, it is suggested that the number 0 will suffice, although this currently causes the analysis icon to be named "Help".

Step 6: Define the Analysis Menu Item and Button:[3] 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 must 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.

3. An alternative way of doing this, which is simpler and more general, needs to be worked out. In the meantime, please consult with the author if necessary.

Step 7: On Demand Code Loading Feature.[4] 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        nil)
     (dialog      nil)
     (covariances nil))
  (load (strcat *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.

6 References

Tierney, L. (1990) *Lisp-Stat: An Object-Oriented Environment for Statistical Computing & Dynamic Graphics*. Addison-Wesley, Reading, Massachusetts.

Young, F.W. (1996) *Vista:The Visual Statistics System.* Research Memorandum 94-1(b). L.L. Thurstone Psychometric Laboratory, Univ. N. Carolina.

4. This feature will probably be changed to take advantage of the autoload feature.