Model-View-Controller Package

Elizabeth Whalen

July 28, 2006

1 Overview

The purpose of the *MVCClass* package is to provide the definitions of classes and generic functions that will be used by other packages to create model-view-controller applications. Currently, the *iSPlot* and the *iSNetwork* packages use the *MVCClass* package to create linked, interactive views of data. The *BioMVCClass* package extends the classes in the *MVCClass* package to include more model and view classes. These definitions were put in a separate R package because the *BioMVCClass* packages depends on the following R packages: *Biobase*, *graph*, and *Rgraphviz*.

2 MVC Class

The `MVC` class represents a model-view-controller object. Basically, this class will bind the model with its views and its controller so that an object of this class can be created each time there is a new data set (model). Thus, there is a one-to-one relationship between a MVC object and a model object. Because of this relationship, the MVC object and the model object are identified by the same name, which is stored in the modelName slot of the model object (see Section 3). Figure 1 shows the inheritance structure for the `MVC` classes.

The `MVC` class is a virtual class that binds the other `MVC` classes together. Every `MVC` object will need the three slots: model, viewList and controller. The model slot stores the model object, the viewList slot is a list of the view objects for this model, and the controller slot is an environment that stores information for this MVC. Inheriting from the `MVC` class is the `singleModelMVC` class that has no new slots. The `singleModelMVC` class represents a `MVC` object that is not linked to any other `MVC` objects.

Inheriting from the `singleModelMVC` class is the `linkedModelMVC` that contains the new slots: parentMVC and childMVCList. The parentMVC slot is the name of the parent model (MVC) and the childMVCList slot is a list of the children models (MVCs). Note that model and MVC objects use the same name so the parentMVC slot refers to both the name of the model and the name of the MVC. Similarly, for the childMVCList slot, the names in the list refer to both the name of the model and the name of the MVC.

Thus, the `singleModelMVC` class binds the model, view, and controller objects together and the `linkedModelMVC` class allows the MVC objects to be related to each through a parent-child (tree) hierarchy because of the slots, parentMVC and childMVCList. These two slots also show that a MVC object can have at most one parent MVC and can have many children MVCs.

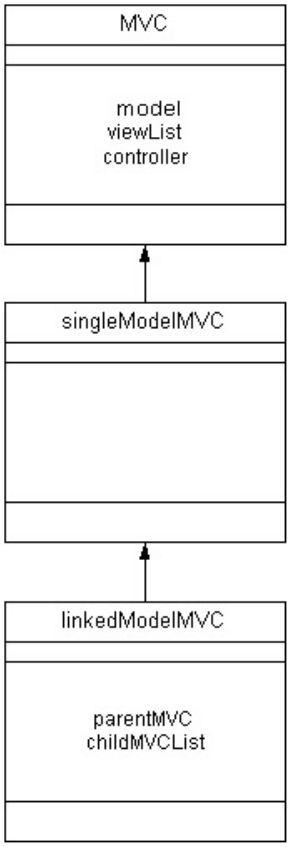

Figure 1: Inheritance for MVC Objects.

3 Model Classes

The model class is responsible for storing and updating the data. These two functions are reflected in Figure 2, which shows the inheritance structure for the model classes. Here the virtual class, `gModel`, has the slots: modelData, linkData, virtualData, modelName, and modelVar. These five slots are common information that all models need. The modelData slot is the data set for this model, the linkData slot is a list of two functions, `toParent` and `fromParent`, which link this model to its parent and child models, respectively (see Section 2), the virtualData slot is data pertaining to the views that needs to be stored with the model so that it can be shown in all views of this model, the modelName slot is the name of the model (and the name of the MVC), and the modelVar slot is a list of named model variables (for example, a statistic for each element in the model).

In this package, the only specific model class is `dfModel`. The `dfModel` class represents a model where the modelData has class data frame (or matrix). Other model classes are defined in the *BioMVCClass* package.

As mentioned previously, the model classes are also responsible for updating the data so a generic function `updateModel` has also been defined. The `updateModel` methods will be defined in packages that use this package (i.e. the methods will be defined in *iSNetwork* and

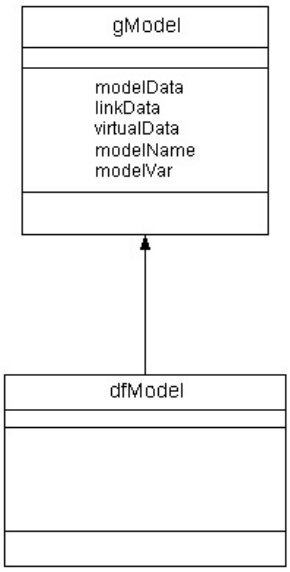

Figure 2: Inheritance for Model Objects.

iSPlot).

The model classes may also be asked to provide information by a `gAskAncestorMessage` object (which is discussed in Section 5) and thus, the generic function `provideInfo` has been defined. Again, the `provideInfo` methods are defined in packages that use this package, such as *iSNetwork*.

4 View Classes

The view classes represent the visual depictions of the model. All views will need to store some common information and respond to certain events through methods. This consideration led to an object model where the different view classes inherit from a view virtual class, called `genView`. The object model for the view classes is shown in Figure 3.

The `genView` class has three slots, dataName, win, and winNum. dataName is the name of the model that the view displays, win is the Gtk window object that holds the view, and winNum is the number of the window so that the window can be identified in the GUI that other packages create. All views will need to know these three pieces of information so `genView` will bind the view classes together.

As for specific view classes, the `spreadView` class will represent a spreadsheet view of the data. This view will only make sense for models that are a two dimensional data structure, such as a matrix or a data frame. The information needed for this view is stored in the slot, clist, which is the Gtk spreadsheet object.

When creating a plot of a data set, there is a general class, called `plotView`, that has the slots, plotDevice, plotPar, and drArea, which store the device number of the plot, the plotting parameters, and the Gtk drawing area object, respectively. This class is a virtual class because it is not intended to have any objects as it just represents a general plot. The plot classes that

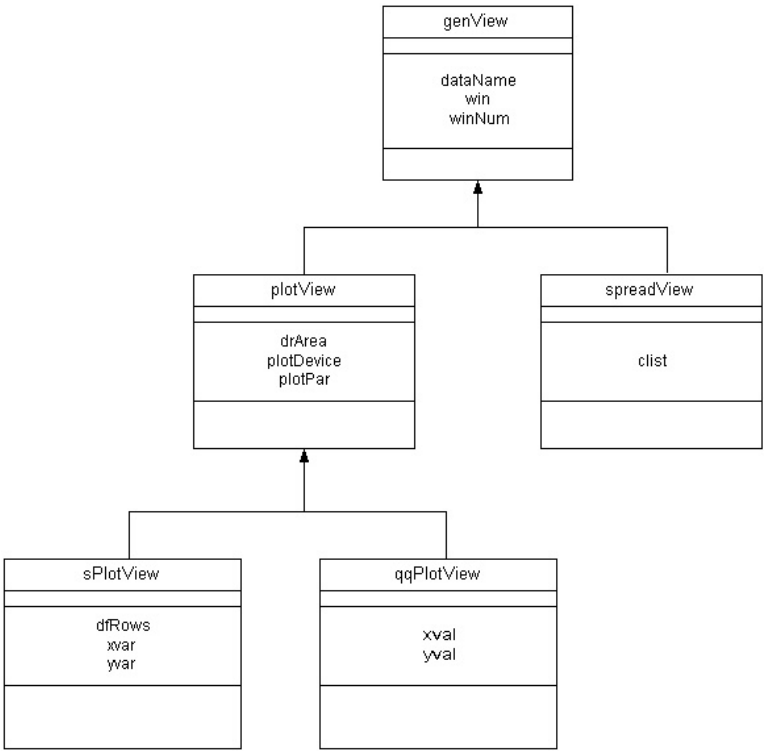

Figure 3: Inheritance for View Objects.

can have objects are the specific plot classes, `sPlotView` and `qqPlotView`, which inherit from the `plotView` class. More view classes are defined in the *BioMVCClass* package.

The `sPlotView` class represents a scatterplot view and it has the extra slots, dfRows, xvar, and yvar, where dfRows are the row names from the model that are shown in the plot, xvar is the variable name (the column name if the model is a data frame) from the model that is shown as the x variable in the plot, and yvar is the variable name (the column name if the model is a data frame) from the model that is shown as the y variable in the plot. The `qqPlotView` represents a qq-plot view and it has the extra slots, xval and yval, where xval are the numeric values of the points on the x-axis and yval are the numeric values of the points on the y-axis.

Several generic functions have been defined that refer to operations performed on views. These generic functions are `motionEvent`, `clickEvent`, `updateView`, `redrawView` and `identifyView`. Methods for these generic functions are defined in packages that use this package, such as *iSPlot* and *iSNetwork*. When these methods are defined, the `motionEvent` method responds to a mouse over event on a view, the `clickEvent` method responds to a click event over a view, the `updateView` method updates only a portion of the view, the `redrawView` method completely redraws the view, and the `identifyView` method identifies an object (such as a point, a node, etc.) on a view.

5 Message Classes

The message classes are intended to provide communication between different components of the MVC design, such as when the model changes a message needs to be sent to the views to let them know that they should be updated. This communication between the model, view and controller is crucial for the pieces to work together and yet, still be independent of each other.

The object model for the message classes was derived from the idea that all messages must have certain methods in common, such as `initialize` and `handleMessage`, because all messages must be created and handled in some manner so that the message is read and acted upon. Also, since messages are sent when something has changed, either through addition, alteration, or deletion, the message classes reflect only certain operations. These common purposes for the messages allow the message object model to be constructed, and this model is shown in Figure 4.

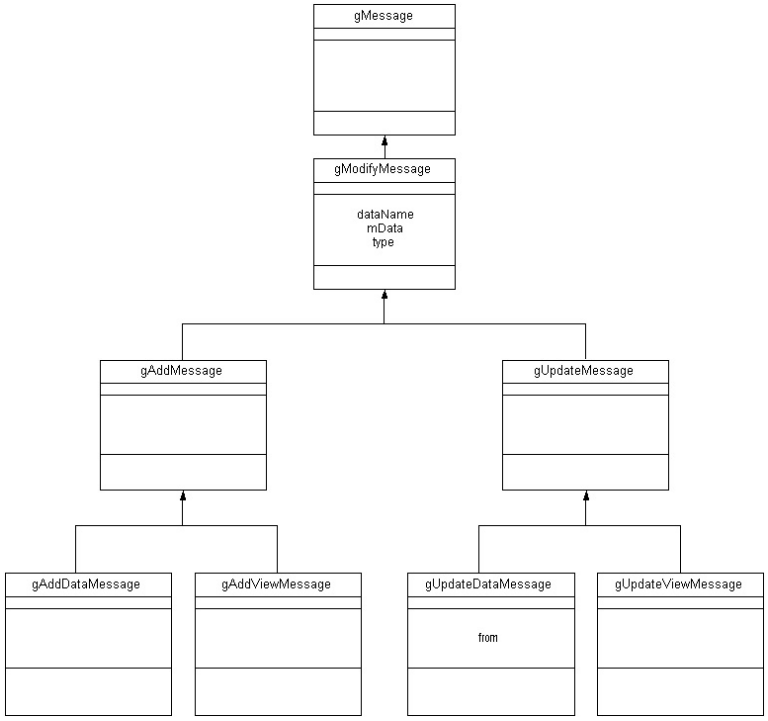

Figure 4: Inheritance for Message Objects.

The figure shows the inheritance structure for message objects that can be passed within one MVC object and Figure 5 in this section shows the inheritance structure for messages objects that can be passed between MVC objects.

As with the view classes, the top message class, `gMessage`, is a virtual class. It contains no slots and its purpose is to bind the other message classes together.

Currently, there are two types of messages for messaging within a model, view, controller object: an add and an update message, and both of these messages are performing some type

of modification. Thus, both the add message, `gAddMessage`, and the update message, `gUpdateMessage`, inherit from the virtual class, `gModifyMessage`, which has three slots, dataName, mData, and type. dataName is the character string that gives the name of the model, mData is a list of data needed to perform the addition or alteration operation, and type is a character string that gives the type of addition or alteration to perform. Both `gAddMessage` and `gUpdateMessage` are also virtual classes.

Since adding a view requires different information and methods from adding a model (and similarly with updating views versus models), it makes sense to have two separate message classes for these elements. Thus, when messages are sent between components, they are from one of the following classes: `gAddDataMessage`, `gAddViewMessage`, `gUpdateDataMessage`, and `gUpdateViewMessage`. As shown in Figure 4, the `gAddDataMessage` and `gAddViewMessage` classes inherit from the `gAddMessage` class, and the `gUpdateDataMessage` and `gUpdateViewMessage` classes inherit from the `gUpdateMessage` class.

The `gAddDataMessage` class represents a message to add a model and a MVC object because whenever a model is added, a MVC object is also added that contains this model. The `gAddViewMessage` class represents a message to create a view.

The `gUpdateDataMessage` class represents a message to update a model. This message is crucial for linking because it is responsible for ensuring that once a model has been updated this information must propagate to any views of the model and this information must also be passed along to any other MVC object that is related to this model. Because this information gets passed to other MVC objects, this message also has one new slot, from, in addition to the slots it inherits from `gUpdateMessage`. The from slot is the name of the model that the update data message came from. This slot is necessary because the update data message can come from one of two sources: the message can occur because of user interaction with a view of this model or the message may occur because a different model that is linked to this model has been updated. So the update data message may come from within this MVC object or it may come from a different MVC object that has changed its model and is related to this MVC object. Information on passing messages between MVC objects is given later in this section. Finally, the `gUpdateViewMessage` class represents a message to update a view.

Message classes have also been defined to pass information between MVC objects. The message types for passing information between `MVCs` are similar to the messages that are passed within `MVC` objects. Now the add child message is similar to the add model message, and the send parent message and the send child message are similar to the update model message. The inheritance structure for messages passed between MVC objects is shown in the Figure 5.

The first message class that will be discussed is `gAddChildMessage`. A `gAddChildMessage` object is very similar to a `gAddDataMessage` object because they are both adding a new model and a new MVC object. The difference with a `gAddChildMessage` object is that it must fill in some extra information to tie this new MVC object to its parent MVC object. The `gAddChildMessage` class inherits from the `gAddMessage` class, with the same slots of dataName, mData, and type. The dataName slot is the name of the new model (and new MVC), the mData slot is a list that contains the model data and virtual data to fill the modelData and virtualData slots, respectively, of the new model, and the type slot gives the type of the model.

The next two message classes pertain to updating MVC objects when a parent MVC or child MVC object has changed. These classes are `gSendParentMessage` and `gSendChildMessage` and they both inherit from the `gMessage` class, as shown in Figure 5. The information these

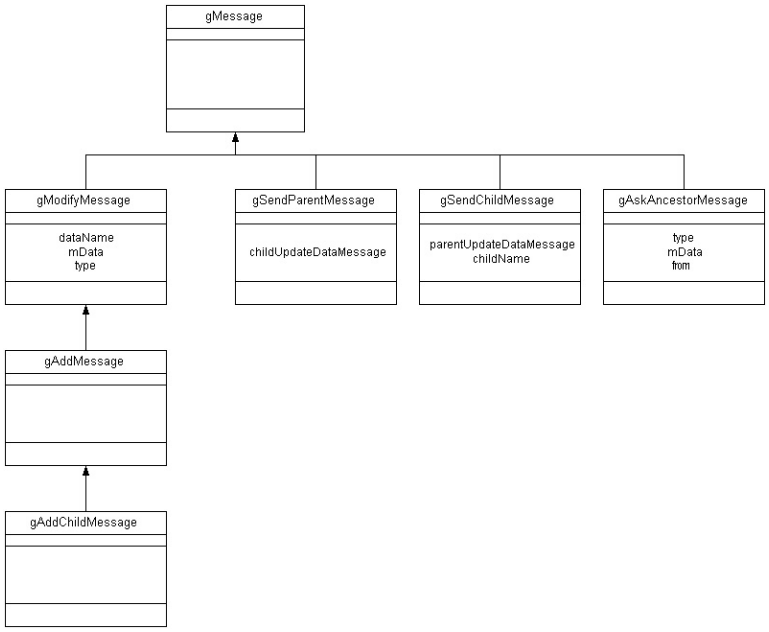

Figure 5: Inheritance for Message Classes that are Passed Between MVCs.

two classes need to contain is a `gUpdateDataMessage` object. Thus, the `gSendParentMessage` class has one slot, childUpdateDataMessage, and this slot contains the `gUpdateDataMessage` object that was used to update the child model. Similarly, the `gSendChildMessage` class has two slots, parentUpdateDataMessage and childName, where the parentUpdateDataMessage slot contains the `gUpdateDataMessage` object that was used to update the parent model and the childName slot contains the name of the child model that is being updated (because a parent MVC can have more than one child MVC).

Finally, the `gAskAncestorMessage` class, which inherits from the `gMessage` class, is used to to ask an ancestor MVC for information about its model. This class is used when a child MVC is being created and it needs information from more than just its immediate parent, i.e. it also needs some information from its grandparent or a different ancestor. The `gAskAncestorMessage` class has three slots, from, type, and mData. The from slot is the name of the MVC that asked for the information (i.e. where this message came from), the type slot is the type of model to look for in an ancestor, and the mData slot is the information being asked for in the model.

A generic function, `handleMessage`, is defined that is used to read message objects. Methods for this generic function are defined in packages that use this package, such as *iSPlot* and *iSNetwork*.

6 Event-Callback Function Class

Linking callback functions to events, such as a button click, a mouse movement or a key press event, is what allows users to have an interactive environment. Then when an event happens

something occurs in response depending on the callback function linked to the event. Now to create a flexible environment we do not want a callback function that always does the same thing when an event occurs. For example, when a left button press event occurs, we may sometimes want a point to be colored and at other times we may want a point to be hidden. To allow this flexibility, a class called `gEventFun` was created. By using this class, users are able to change the response to an event.

The `gEventFun` class stores all of the information about a potential callback function. Having the callback function information stored in an object allows a user to connect events to callback functions. The class definition for `gEventFun` is shown in Figure 6.

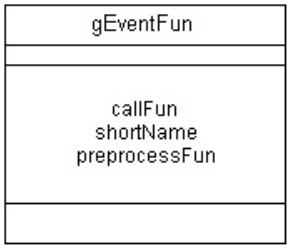

Figure 6: The gEventFun Class.

As shown in Fig 6, the slots in the `gEventFun` class are callFun, shortName, and preprocessFun. The callFun slot is the callback function, the shortName slot is a short description of the function, and the preprocessFun slot is a list of all preprocessing functions that must be called before the callback function.

The callback function that is stored in the callFun slot must take only one parameter and the class of this parameter depends on the model class in the active MVC object. For example, for a data frame model, the parameter is of class character, representing a row in the data frame.

The preprocessing functions that are stored in the preprocessFun slot take no parameters. The purpose of the preprocessing functions is to set some variables that the callback function needs. Thus, not all callback functions need preprocessing functions and if no preprocessing functions are needed, then this slot is set to NULL. If the callback function sets the color of a node in a graph, then a preprocessing function is needed to determine what color is used. So in this example, the preprocessing function opens a color browser to allow the user to choose what the new color is.

7 Conclusions

The inheritance structure for the classes defined in this package are intended to be extensible. If a user needs a model of a different type that is not currently represented, then a new model class can be defined that inherits from the `gModel` virtual class. Similarly, new view and message classes can be created if new views or messages are needed by a user.

As mentioned in Section 1 this package is intended to be a behind the scenes package that is used by other packages for its class and generic function definitions. Currently, this package is used by the *iSPlot* and *iSNetwork* packages, and is extended by the *BioMVCClass* package.