Chapter 1

The model-view-controller pattern

The model-view-controller design pattern for GUIs (MVC) is a means to isolate the data, the graphical represtation of the data, and the code that connects the two. The data is stored in a *model*, this data may be represented by one or more *views*, and a *controller* connects a model with a view.

According to a wikipedia article

> MVC was first described in 1979 by Trygve Reenskaug, then working on Smalltalk at Xerox PARC. The original implementation is described in depth in the influential paper "Applications Programming in Smalltalk-80: How to use ModelâĂŞViewâĂŞController".

It is widely implemented. In `RGtk2` some of the more complicated widgets, such as the tree view, a text view etc., have an explicit specification of the model. In `tcltk`, the TCL variables play the role the model and the TK widgets are views of the model.

In `R` an implementation is given in the bioconductor pacakge `MVCClass` using S4 classes. In a subsequent chapter, we provide a lightweight implementation using the `proto` package.

A common use, found in the `ggobi` package say, is graphical brushing. That is, when a user identfies certain points in one graph displaying a data set, the same points are highlighted in a different graph of the same data set.

In MVC language, we have the model is the data set, and there are two views – the two graphs. When one graph is brushed a controller informs the model that there is a change in selection. The model then notifies any view that the selection has changed (again through a controller) and the view updates its representation accordingly. By inserting a controller between the model and the view, the two are decoupled which provides several benefits. These benefits include, the overall program flow is easier to debug, the decoupling allows views and models to be reused, and the model can be changed independent of the view (so one can update the graphical displays without having to remake the graphs). Of course, the benefits come at a cost – increased complexity, atleast conceptually.

1.1 A basic implementation

To illustrate the concept and the responsibilities for each part we present an implementation using the `proto` package. This package extends R's environments to create a somewhat object-oriented programming style.

Our implementation follows somewhat that given in the `pygtkmvc` python package (`http://sourceforge.net/apps/trac/pygtkmvc/wiki`). There are many different implementations, this one is relatively straightforward.

A base trait

The `proto` package implement protoype programming which is not technically object-oriented, but does allow for the main features: a concept of object properties, object methods, and object inheritance. However, the concept of a class is not used. (The `mutatr` package does something similar.)

Instead of classes, one defines a "trait" that provides the standard properties and methods for an instance. Sub traits can inherits these and modify them, as we will illustrate.

We load two package to start.

```
require(proto)
require(digest)
```

We begin by defining a base trait from which our `Model`, `View` and `Controller` traits will inherit. The naming convention is uppercase camel case.

```
BaseTrait <- proto()
```

The `proto` package allows for properties and methods to be defined within the `proto` call, but for typesetting reasons we will assign using the `$` notation for environments.

We define a class property to keep track of the type of `proto` object we have, as there is not built in class concept.

```
BaseTrait$class <- "Base"
```

Defining a property, as above, is straightforward. Defining a method is a bit different, as the functions have an initial argument of `.`. This allows the object to be passed to the function body. As `proto` objects are just environments, they are mutatable, so when the assignment is made in this method definition, the value of `class` property is updated in the object outside the function body.

```
BaseTrait$add_class <- function(., newclass) .$class <- c(newclass, .$class)
```

We define a simple method to append a value to a property which is storing a list of values.

```
BaseTrait$append <- function(., name, value, key) {
  val <- get(name, envir=.)
```

```
  if(is.list(val)) {
    if(!missing(key))
      val[[key]] <- value
    else
      val[[length(val)+1]] <- value
  } else {
    val <- c(val, value)
  }
  assign(name, val, envir=.)
}
```

The proto package allows for introspection – determining properties at run time. We implement some methods for doing so. First, an implementation mirroring the is function of S4 programming.

```
BaseTrait$is <- function(., class=NULL) {
  if(!is.null(class))
    class %in% .$class
  else
    TRUE
}
```

This function returns all properties and methods. The trick is that children of a proto object inherit methods and properties, but ls does not list those, so we walk backwards using parent.env (called in a OO manner).

```
BaseTrait$list_objects <- function(., class=NULL) {
  s <- .
  if(!s$is(class))
    return(list())
  out <- ls(s, all.names=TRUE)
  while(is.proto(s <- s$parent.env())) {
    if(s$is(class))
      out <- c(out, ls(s, all.names=TRUE))
  }
  unique(out)
}
```

This method returns all methods of a certain class, as defined by the class property (not the class in the S3 or S4 sense).

```
BaseTrait$list_by_class <- function(., class) {
  out <- .$list_objects()
  out <- sapply(out, function(i) {
    obj <- get(i, envir=.)
    if(is.proto(obj) && obj$is(class))
      obj
  })
  out[!sapply(out, is.null)]
}
```

Next two methods to list just the properties and the methods.

```
BaseTrait$list_properties <- function(., return_names=FALSE, class=NULL) {
  ## will return objects or just names if return_names=TRUE
  ## can pass class= value if desired
  out <- .$list_objects(class=class)
  out <- sapply(out, function(i) {
    ## skip "class" and dot names
    if(i != "class" && !grepl("^\\.",i) && !grepl("^\\.doc_",i)) {
      obj <- get(i, envir=.)
      if(!is.function(obj))
        obj
    } else {
      NULL
    }
  })
  out <- out[!sapply(out, is.null)]
  if(return_names)
    names(out)
  else
    out
}
```

```
BaseTrait$list_methods <- function(.) {
  nms <- .$list_objects()
  ind <- sapply(nms, function(i) {
    is.function(get(i,envir=.))
  })
  nms[ind]
}
```

Finally, we make a convenience function to call a method provided that method exists and is a function.

```
BaseTrait$do_call <- function(., fun, lst=list()) {
  if(exists(fun, envir=.) && is.function(FUN <- get(fun, envir=.)))
    do.call(FUN, c(., lst))
}
```

A model trait

A model consists of properties and methods to manipulate these properties. In addition we implement the observer pattern. An observer is a controller. When a model property changes, all the observers are notified of this. The observer can then update any views it is associated with. In order to implement this, we must change the property values through the `setattr` method. For convenience, the `init` method will create `get/set` pairs for interacting with the property values by name.

```
Model <- BaseTrait$proto()
Model$add_class("Model")
```

We implement the observer pattern by defining a few key methods. An oberver is a controller instance (defined later). The following just stores these in a list using a private property.

```
Model$.observers = list()              # private property (leading .)
Model$add_observer <- function(., observer) {
  if(is.proto(observer) && observer$is("Controller")) {
    id <- length(.$.observers) + 1
    .$.observers[[id]] <- observer
  }
}
```

The `proto` package provides the `identical` method to compare two proto objects. We use this to remove an observer when requested.

```
Model$remove_observer <- function(., observer) {
  if(!missing(observer) && (is.proto(observer) && observer$is("Controller"))) {
    ind <- sapply(.$.observers, function(i) i$identical(observer))
    if(any(ind))
      sapply(which(ind), function(i) .$.observers[[i]] <- NULL)
  }
}
```

This is the key method, which is called when a property value is changed through `setattr`. The controllers use a naming convention. If a property `prop1` is changed, then the methods `property_prop1_value_changed` and `model_value_changed`, if present in the controller, are called.

```
Model$notify_observers <- function(., key=NULL, value=NA, old_value=NA) {
  sapply(.$.observers, function(i) {
    if(digest(value) != digest(old_value)) { #serialize, then compare
      if(!is.null(key)) {
        i$do_call(sprintf("property_%s_value_changed",key),
                  list(value=value, old_value=old_value))
      }
      i$do_call("model_value_changed", list()) # always call if present
    }
  })
  invisible()
}
```

The methods `getattr` and `setattr` are used to interact with the model's properties,

```
Model$getattr = function(., key) get(key, envir=.)
Model$setattr = function(., key, value) {
```

```
  old <- .$getattr(key)
  assign(key, value, envir=.)
  .$notify_observers(key=key, value=value, old_value=old)
}
```

On initialization, a model has get/set methods defined for its properties, as a convenience to using `getattr` and `setattr`.

```
Model$init = function(.) {
  sapply(.$list_properties(return_names=TRUE), function(i) {
    assign(paste("get_",i, sep=""),
           function(.,...) .$getattr(i),
           envir=.)
    assign(paste("set_", i, sep=""),
           function(., value, ...) .$setattr(i,value),
           envir=.)
  })
  invisible()
}
```

A view trait

A view typically provides a visual representation of a model property or properties. (Not all case, as a model could also be a view, etc..) Our view trait is oriented around using `gWidgets` to provide the graphical widgets.

```
View <- BaseTrait$proto()
View$add_class("View")
```

The basic view properties include a list of attributes to pass to the widget constructor (`make_ui`) and a list of widgets, to which we provide a few convenience methods.

```
View$attr <- list()                    # passed to widget constructor
View$widgets <- list()                      # lists all widgets in the view
View$get_widgets <- function(.) .$widgets
View$get_widget_by_name <- function(., key) .$get_widgets()[[key]]
```

The user interface for a view is created by the `make_ui` method. This should create the widgets and save those that will be referenced later in the `widgets` property.

```
View$make_ui <- function(., cont, attr=.$attr) {}
```

The view has two distinct states – before the widget is realized and after. It is important to be able to determine which state the widget is in.

```
View$is_realized <- function(.)
  length(.$get_widgets()) && isExtant(.$get_widgets()[[1]])
```

Communication between the model and view is done through the controller. These methods are there to provide a standard interface in the simplest cases. These just provide a convenient means for the controller, they do not sychronize with the model, as the view does not know the model or even the controller.

```
get_value_from_view = function(.) {}
set_value_in_view = function(., widget_name, value) {
  if(.$is_realized()) {
    widget <- .$get_widget_by_name(widget_name)
    svalue(widget) <- value
  }
}
```

Finally, we define similar functions to hide or disable the view.

```
View$enabled <- function(., bool)
  if(.$is_realized())
  invisible(sapply(.$get_widgets(), function(i) enabled(i) <- bool))
View$visible <- function(., bool)
  if(.$is_realized())
  invisible(sapply(.$get_widgets(), function(i) visible(i) <- bool))
```

A controller trait

Controllers have the difficult task of implementing the core logic that connects the various views and models. A controller needs to know the model and the view and provide a means for the two to communicate back and forth, if desired.

```
Controller <- BaseTrait$proto()
Controller$add_class("Controller")
```

We first define a `model` property and some methods to interact with it. When setting the model, we also take care to update the observers including the `adapters` which are a simple form of a controller discussed later.

```
Controller$model <- NULL
Controller$get_model <- function(.) .$model
Controller$set_model <- function(., model) {
  if(is.proto(model) && model$is("Model")) {
    .$model$remove_observer(.)
    .$model <- model
    .$model$add_observer(.)
    sapply(.$.adapters, function(i) i$set_model(model))
  }
}
```

Similarly, we define a `view` property and some methods to interact with it.

```
Controller$view <- NULL
Controller$get_view <- function(.) .$view
Controller$set_view <- function(., view) {
  if(is.proto(view) && view$is("View")) {
    if(is.proto(.$get_view()) && .$get_view()$is("View"))
      .$remove_view()
    .$view <- view
    sapply(.$.adapters, function(i) i$set_view(view))
```

```
  }
}
Controller$remove_view <- function(.)
  sapply(.$.adapters, function(i) i$remove_view())
```

These methods are used in the definition of the adapter pattern given later. They are used to synchronize changes in the model with the view and vice versa.

```
Controller$update_from_model <- function(.) {}
Controller$update_from_view <- function(.) {}
```

This initialization method is used to connect the controller to the model (as an observer) and to propogate the model values to the view the initial time.

```
Controller$init <- function(.) {
  if(!is.null(.$get_model())) {
    .$update_from_model()
    .$get_model()$init()
    .$get_model()$add_observer(.)
  }
  if(!is.null(.$get_view())) {
    .$update_from_view()
  }
  .$register_adapters()
  ## call value_changed methods to update any views
  nms <- .$list_methods()
  sapply(nms[grep("property_(.*)_value_changed$", nms)],
         function(i) {
           prop <-  gsub("property_(.*)_value_changed$","\\1",i)
           get(i, envir=.)(., .$get_model()$getattr(prop),NA)
         })
  invisible()
}
```

The controller is used as an observer for its model. Sub classes may override methods such as these to implement specific actions when the model is changed. These follow the naming convention needed by our implementation of the observer pattern.

```
Controller$model_value_changed <- function(.) {}

Controller$property_PROPERTYNAME_value_changed <- function(., value, old_value) {}
```

Defining a controller can be a bit involved. The adapter pattern simplifies this for the simple case that a single property is being observed and the view has just a single widget to update.

We define an adapter using a list of lists:

```
## list of adapters. Each adapter specified with a list. E.g.,
## list(property="propname",
##      view_name="viewname",
```

```
##      add_handler_name=c("addHandlerChanged"), # or NULL to suppress
##      handler_user_data=NULL
##      )
Controller$adapters <- list()
```

The adapters are constructed by the `register_adapters` method which is called in the `init` method. The actual adapter instances are stored in this private property.

```
Controller$.adapters <- list()
Controller$.handlerIDs <- list()
```

Finally, our method to register the adapters is defined using the `Adapter` trait given below.

```
Controller$register_adapters <- function(.) {
  if(length(.$adapters) && !length(.$.adapters)) {
    .$.adapters <- lapply(.$adapters, function(i) {
      Adapter$proto(model=.$get_model(),
                    view=.$get_view(),
                    property=i$property,
                    view_widget_name=i$view_widget_name,
                    add_handler_name=i$add_handler_name,
                    handler_user_data=i$handler_user_data
                    )
    })
  }
  if(length(.$.adapters))
                                    sapply(.$.adapters, function(i) i$init())
}
```

Next we define the adapter trait.

```
Adapter <- Controller$proto()
Adapter$add_class("Adapter")
```

We define some properties of the adapter. We specify the model property and name of the widget in the view for starters.

```
Adapter$property <- NULL
Adapter$view_widget_name <- NULL       # otherwise last one
```

The view communicates back to the model through the controller through a callback. This defines the `gWidgets` "`addHandlerXXX`" to be used. One can leave this an empty string for no interaction

```
Adapter$add_handler_name <- c("addHandlerChanged") # 1 or more
Adapter$handler_user_data=NULL
```

This method is called by `init` to add a moel observer that updates the widget value.

```
Adapter$update_from_model = function(.) {
  ## set up model to notify view For example:
  view <- .$get_view()
  meth_name<- sprintf("property_%s_value_changed", .$property)
```

```
  assign(meth_name,
         function(., value, old_value) {
           view$set_value_in_view(.$view_widget_name, value)
         },
         envir = .)
  ## call method
  get(meth_name, envir=.)(., .$get_model()$getattr(.$property), NA)
}
```

This method is called by `init` to add handlers to the widget to propogate changes back to the model.

```
Adapter$update_from_view <- function(.) {
  ## here view knows about model through controller (this adapter)
  if(!.$get_view()$is_realized()) return()
  if(!is.null(.$view_widget_name))
    widget <- .$get_view()$get_widget_by_name(.$view_widget_name)
  else
    widget <- tail(.$get_view()$get_widgets(), n=1)[[1]]
  ## gWidgets specific call to set up control between model and
  ## view
  if(is.null(.$add_handler_name))
    .$add_handler_name="addHandlerChanged"
  sapply(.$add_handler_name, function(i) {
    if(i != "") {
      lst <- list(obj=widget,
                  handler=function(h,...) {
                    . <- h$action$adapter

                    ## set property in model using name
                    value <- svalue(h$obj)
                    if(isExtant(h$obj)) {
                      .$model$setattr(.$property, value)
                    }
                  },
                  action=list(adapter=.))
      .$append(".handlerIDs", do.call(i, lst))
    }
  })
}
```

This method is called to remove a view, disconnecting the handlers that have been defined first.

```
Adapter$remove_view <- function(.) {
  if(exists(".handlerIDs", .))
    sapply(.$.handlerIDs, function() removeHandler(.$get_view(), i))
}
```

This initialization method sets up the adapter.

```
Adapter$init <- function(.) {
  ## check that we are all there
  if(!is.null(.$property) &&
     (is.proto(model <- .$get_model()) && model$is("Model")) &&
     (is.proto(view <- .$get_view()) && view$is("View"))) {
    .$update_from_model()
    .$update_from_view()
  } else {
    warning("Adapter does not have view, model and property")
  }
  .$model$add_observer(.)
}
```

Examples

Silly example using an adapter Our first example uses a model with just two properties.

```
model <- Model$proto(prop1=1, prop2="button label")
```

We still need to initialize this model if the get/set pairs are desired.

Our view, for sake of illustration, has a text area and a button.

```
require(gWidgets)
options(guiToolkit="RGtk2")
view <- View$proto(make_ui=function(., cont, attr=.attr) {
  .$widgets[['toplevel']] <- w <- gwindow("Example")
  g <- ggroup(cont=w, horizontal=FALSE)
  .$widgets[["text"]] <- gtext("", cont=g)
  .$widgets[["button"]] <- gbutton("button", cont = g)
})
```

We use an adapter to connect the text widget with the first property and the button widget with the second. As we don't modify the model when the button is clicked, we specify the handler name with an empty string.

```
adapter <- Controller$proto(model=model, view=view,
                              ## call and adapter
                              adapters=list(
                                prop=list(
                                  property="prop1",
                                  view_widget_name="text"
                                  ),
                                ## button doesn't need to set model
                                button=list(property="prop2",
                                  view_widget_name="button",
                                  add_handler_name="")
                                )
```

```
                                )
model$init()
view$make_ui()
adapter$init()
```

To do the same thing with a controller is a bit more involved. We woul override the methods `update_from_model` and `update_from_view` which set up the communication between the model and the view.

```
controller <- Controller$proto(model=model, view=view)

controller$update_from_model <- function(.) {
  .$property_prop1_value_changed <- function(., value, old_value)
    svalue(.$view$widgets[['text']]) <- value
  .$property_prop2_value_changed <- function(., value, old_value)  {
    button <- .$view$get_widget_by_name("button")
    svalue(button) <- value
  }
}
```

This defines communication between the text entry and the model. No action is given to the button, although in practice this wouldn't make any sense.

```
controller$update_from_view <- function(.) {
  widget <- .$get_view()$get_widget_by_name("text")
  .$append(".handlerIDs", addHandlerChanged(widget, handler = function(h,...) {
    . <- h$action$controller
    model <- .$get_model()
    val <- svalue(h$obj)
    model$set_prop1(val)
  }, action=list(controller=.))
           )
}
```

One model property two views Our next example shows how we can share a model among views, in this case to sychronize a spinbutton and a slider widget.

```
model <- Model$proto(value=1)
model$init()

view <- View$proto()
view$make_ui <- function(.) {
  .$widgets[['toplevel']] <- (w <- gwindow("Example"))
  g <- ggroup(cont = w, horizontal=TRUE, expand=TRUE)
  .$widgets[['slider']] <- gslider(from=0, to=10, by=1, cont=g, expand=TRUE)
  .$widgets[['spinner']] <- gspinbutton(from=0, to=10, by=1, cont=g)
```

```
}
view$make_ui()

controller <- Controller$proto(model=model, view=view)
```

We bypass `update_from_model` and define the value changed method directly:

```
controller$property_value_value_changed <- function(., value, old_value) {
  ## update both widget
  widgets <- lapply(c("slider","spinner"), function(i)
                    .$view$get_widget_by_name(i))
  sapply(widgets, function(i) svalue(i) <- value)
}
```

The same handler is used for each widget to update the model, giving us the following code to add the callbacks when the GUI is updated by the user.

```
controller$update_from_view <- function(.) {
  handler <- function(h,...) {
    . <- h$action$controller
    model <- .$get_model()
    val <- svalue(h$obj)
    model$set_value(val)
  }
  widgets <- lapply(c("slider","spinner"), function(i)
                    .$view$get_widget_by_name(i))
  IDs <- sapply(widgets, function(i) addHandlerChanged(i, handler=handler, action=list(controlle
}
controller$init()
```