Chapter 7: Interactive System Development Framework

So far, we have only focused on the UI objects and their behavior. Obviously a whole application is made up of not only UI objects but those for the core functions of the application as well. How do we effectively develop the larger interactive programs with such two parts (UI and internal functional core)? For this, it is a good idea to follow an established "development" framework or methodology suited for highly interactive systems. A **development framework** refers to a modular approach for interactive program development where the core computational and interface parts are developed in a modularized fashion and combined in a flexible manner. Such a development framework is often much based on the UI toolkit which provides the abstraction for the interface parts. For one, the framework allows the concept of plugging in different interfaces for the same model computation and easier maintenance of the overall program. In addition, such a practice also promotes the productivity and easier and less costly post-maintenance. MVC (Model, View and Controller) is one such major framework.

7.1 Model, View and Controller (MVC)

The MVC approach was first proposed as a computational architecture for interactive programs (rather than a methodology) by the designers of the programming language called SmallTalk which is one of the first object-oriented and modular languages [2]. The modular nature of the MVC architecture naturally shaped the interactive program development style or methodology. With the MVC framework, the application is divided into three parts: (1) Model, (2) View and (3)

Controller as illustrated in Figure 7.1.

[Figure 7.1] The Model-View-Controller architecture for interactive applications.

7.1.1 Model

The "Model" part of the application corresponds to the computation (e.g. realized as objects) that deals with the underlying problem or main information or data of the application. For all practical purposes, once in place, "Model" of the application tends to be stable and unchanging. For instance, in an interactive banking application, the "Model" will be parts of the program that maintain the balance, compute the interest, make wire transfers and etc. The Model has no knowledge of how the central information will be presented to the user (output/presentation), or how the transactions (input) are handled.

7.1.2 View

The "View" part of the application corresponds to the implementation for output and presentation of data. In modern GUI based interfaces, the implementation will typically be consisted of widgets. For instance, views might be windows and widgets that display the list of transactions and balance of a given account in a banking application, or playing of a background audio clip depending on the score level for a game. As a whole, there may exist multiple views for a single

application (or model). For instance, there could be different view implementations for different display platforms or user groups (e.g. 17 inch monitor, 10 inch LCD, HD resolution display, display with vibro-tactile output device, young users, elderly users). Note that the output display does not necessarily have to be visual.

Anytime, the model is changed, the view of that model must be **"notified"** so that it can change the visual representation of the model on the output display. Region/portion of the screen/display that is no longer consistent with the model is called "damaged." Often times, it is be too tedious to update just the damaged part of the display upon change of information in the model. The practical approach is to redraw the entire content of the "smallest" widget that encompasses the damaged region or redrawing the entire window.

7.1.3 Controller

The "Controller" part of the application corresponds to the implementation for manipulating the view (in order to ultimately manipulate the internal model). It takes external inputs from the user, interprets and relays them to the model. Controller thus practically takes care of the input part of the interaction. It uses the underlying UI execution framework or operating system to achieve this purpose (while the "View" is mostly independent from the operating system or platform).

In Chapter 6, we studied the mechanism of the UI execution framework in terms of how it identifies and maps the raw user input to the object in focus. In order to find the object in focus (which visual object is to be manipulated on behalf of the model), the controller must

communicate with the view objects. In addition, the controller might also change the content of the display without changing the model sometimes. For instance, if the user wanted to simply change the color of a button (e.g. for UI customization purpose), the controller can directly communicate with the view to achieve this effect.

Once the object in focus is identified, the corresponding event handler would be invoked. The controller will only **"relay" a query** or message for certain change or manipulation to happen to the model rather than making the change itself.

7.1.4 View/Controller

In many application architectures, the view and controller may be merged into one module or object, because they are so tightly related to each other. For instance, a UI button object will be defined by attribute parameters such as its size, label, color, and also the event handler which invokes the methods on the model for change or manipulation.

The MVC architecture or development methodology makes it much easier, particularly for large scale systems, quickly explore and implement and modify various different user interfaces (view/controller) for the same core functional model. This is based on the famous software engineering principle, the separation of concern.

7. 2 Example of MVC Implementation 1: Simple Bank Application

We illustrate a very simple object oriented implementation for an interactive banking application.

In this simple application, the model maintains the balance for a user who can make deposits or withdrawals through a computer user interface. Figure 7.2 shows the overall structure of the application according to the MVC architecture.

[Figure 7.2] An overall MVC based implementation (class diagram) for a simple interactive banking application.

In the figure, as for the "Model" part, a class called "Account" maintains the customer name, balance and two view/controllers (one for displaying the balance and realizing the UI for making deposits, and the other for withdrawal). The model has two core methods for maintaining the correct balance when a deposit (*Account::Deposit*) or withdrawal (*Account::Withdrawal*) is made. These two methods use the *Notify_depositVC* and *Notify_withdrawalVC* to notify the corresponding view for updating the balance in the display.

In this particular example, the View and Controller parts are merged into one class, called the *AccountViewController* which has a pointer to the corresponding model object (as the recipient of the notifications and model change queries). This class is a subclass of a more general *UIObject* that is capable of housing constituent widgets and reactive behavior to external input. It is also the superclass for subclasses, the "*DespositViewController*" and "*WithdrawalViewController*" which

implement the two views/controllers for the given model.

[Figure 7.3] Methods for the class *Account* (the Model).

The subclasses, among others, implement three important virtual methods, "*Init_ui_display*," "*Update_ui_display*," and "*Handle_ui_event*" (Figure 7.3). Each of them are responsible for creating and initializing the display and UI objects within the view/controller, updating the display (invoked by the notification method from the model, see Figure X), and handling the user input. Figure 7.4 shows the "*Handle_ui_event*" method of the *DepositViewCotnroller* which interprets the user input (e.g. textual input of digits into integers) and invokes the model method to e.g. make a deposit by calling *my_model->Deposite(deposit_amount)*. Understand that this will eventually change the model and the view/controller will be notified to change its display (e.g. to show the proper amount of balance after the deposit). Although not shown, the "*WithdrawalViewController*" would be coded in a similar manner.

[Figure 7.4] The *DepositViewController* class and its method, *Handle_ui_event*.

7.3 Example of MVC Implementation 2: No Sheets

As a second example, we will illustrate parts of the implementation code for the "No Sheets" application introduced in Chapter 4 as shown in Figure 7.5. The core of the model is the music information, a list based data structure that contains the chord information of a music read from a user selected file. Aside from the music information itself, there may be other model variables such as the music file name, tempo value, etc. Thus, the model information is updated by and read from the view/controller objects.

The view/controller is composed of several "Activity" (screen interface) objects. The "*SmartChordActivity*" represents the main front end interface which allows the user to apply certain major actions such as selecting/loading the music file, selecting the tempo, playing the chosen music file, and other miscellaneous functions. It will access information from the model, for instance, the name of the current file and current tempo and show them in the interface. "*FileActivity*" represents the file selection interface screen which presents the user with a list of available music files. The user makes a selection and the "*FileActivity*" will construct the internal chord event list and update the model. Likewise, "*TempoActivity*" allows the user to select the tempo and update the model accordingly. Finally, the "*PlayActivity*" accesses the event list data structure of the model and presents the musical information at a given tempo (no model updating is carried out).

[Figure 7.5] The MVC based program structure for the "No Sheets" application introduced in Chapter 4.

7.4 Summary

In this chapter, we have studied one interactive application development methodology called the MVC. MVC is based on the principle of the separation between the UI and core computational functionalities of a given application. Such a separation of concerns allows for the two to be mixed and matched (for exploring different combinations of a proper set of functions and corresponding UIs) and lends itself to easier code maintenance. However, sometimes, it is not very clear whether a given application can be cleanly separated into two parts, namely, the core function and UI. For example, suppose one is to implement several different "Views" for different user groups for the same banking application, and yet another view for changing and selecting the views themselves. In this situation, it seems that the "change of view" functionality is one of the core functions and feature of the application, yet in theory, the "view change" seems to belong to the "View" rather than the "Model."

References

[1] Olsen, Dan, *Developing user interfaces – interactive technologies.* Morgan Kaufman, 1998.

[2] Krasner, Glenn E.; Stephen T. Pope (Aug?Sep 1988). "A cookbook for using the model-view controller user interface paradigm in Smalltalk-80". The JOT (SIGS Publications). Also published as "A Description of the Model-View-Controller User Interface Paradigm in the Smalltalk-80 System" (Report), ParcPlace Systems; Retrieved 2012-06-05.