THE MODEL-VIEW-CONTROLLER STRUCTURE

Introduction

In the course so far we have been trying to design graphical programs according to a certain pattern, namely separating the models of our objects (for example a house, circle or a car) and the different viewer classes. We will develop these thoughts even further and try to make our programs into classes with different ”roles”: model, view, and controller.

- The *Model* stores the state of the application and supplies methods to update (change, add, or remove data) and read (parts of) the state.
- The *View* creates the display and presents the current state of the model, via a reference to the model and the methods to read the state from the model.
- The *Controller* interacts with both the view and the model; it responds to user requests, updates the model and informs the view that it needs to be repainted.

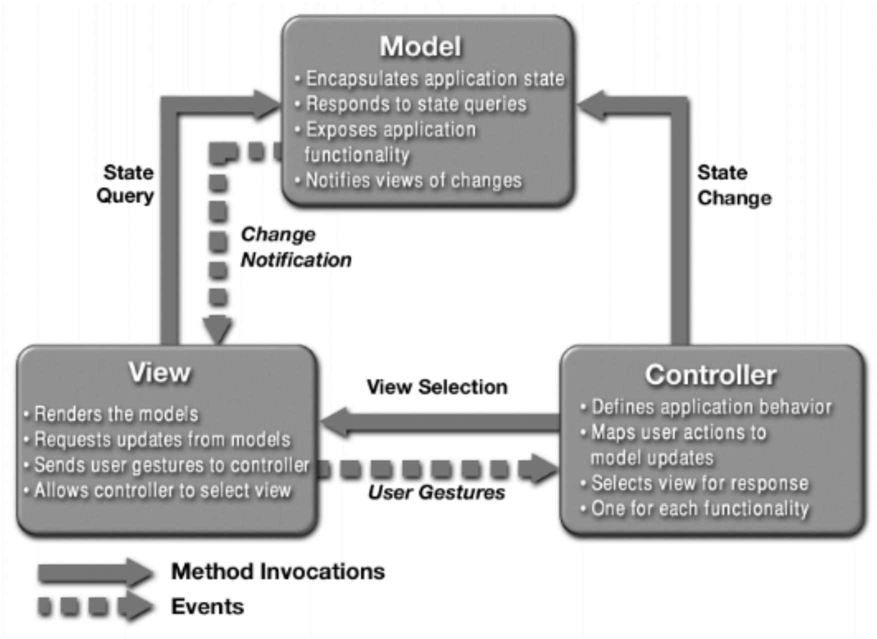
Model
• Encapsulates application state
• Responds to state queries
• Exposes application functionality
• Notifies views of changes
State Query
State Change
Change Notification
View
• Renders the models
• Requests updates from models
• Sends user gestures to controller
• Allows controller to select view
View Selection
User Gestures
Controller
• Defines application behavior
• Maps user actions to model updates
• Selects view for response
• One for each functionality
Method Invocations
Events

Figure 1. The MVC architecture

You might think that this sounds quite vague, but MVC is an example of a design pattern, i.e. an idea that can be reused and applied to many different kind of problems. It is also worth noticing that the MVC structure sometimes is referred to as an architecture. Just as it takes careful planning to construct and build a house, we need to plan our programs before we start coding!

The MVC structure can be useful in many different applications and you will benefit from using it in several different ways:

(1) If you are new to programming MVC will give you a suggestion on how to organize your program into classes. In trying to identify the model and its methods you will acquire insight into the problem. It is very useful to separate the model from details concerning the presentation to the user.

(2) The model is not dependent on its presentation to the user which means that

   - the model will be unchanged, or just slightly changed when switching to a new user interface. Through experience you will find out that it is the presentation of the model that changes the most during the life span of a program.
   - you can have several different views presenting the same data (i.e. the same model). For example you can present numbers as a table, a bar chart or pie chart. Another version is that you can either have a GUI or a command line interface to your program.

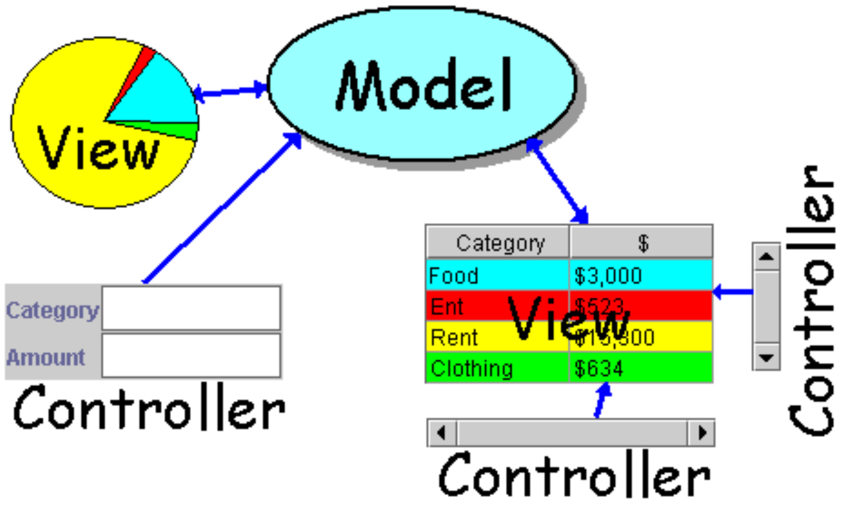

Figure 2. Several views

(3) The view is not aware of the difficult calculations that may take place in the methods that update the model. It is only concerned with the information it needs to

visualize the model. This will simplify the code in both the viewer and the model classes.

(4) The controlling structure will be more obvious as we in the controller class will see how different user actions will update the model.

(5) If you are interested in learning about the more advanced Swing components like `JTable` and `JTree`, you will benefit from MVC since these classes have separate model classes.

However, many of the components in Swing are based on a simplified version of MVC, where the model is separate, but the view and controller have been merged (into what is called a delegate). Not so surprising since the purpose of these classes is to provide a visual controller. Sometimes it is not possible to separate the controller and the view, which makes the simplified version of MVC useful.

Guide to Remodeling Balls in a box according to MVC

Step One.

- Begin with the class `Ball.java`. Make the class into a pure model of the ball object and remove any references to its container (the box). Also remove how the ball is visually rendered.
- Make the class `Box.java` into a second model class. Remove any methods connected to the visualization of the box (do you need the class to inherit from `JComponent`?).
- Create a viewer class to visualize the box on the screen. Make sure the class has a reference to a box object from which you can extract the necessary information to draw the balls.
- Create a control class which holds references to both the viewer object and the box object. Think about what methods fit in this class, how about the method that moves all balls?
- Make a ”Main” class that creates balls and creates objects from the box, viewer and control classes.

Step Two. We will now expand the program so that it can communicate with the user. We want to add buttons to the program so that the user can move balls (one at a time or all the balls several steps). The *events* resulting from the user using the buttons need to be handled. To each *event source* you need to add an *event listener*. Each listener object is an instance of a class that implements the `ActionListener` interface (see pp 355-362 in the book). I have made a small example which you can download (from my web site). It is called `SmallExample.zip` and can be good to look at before you start implementing the modifications to the program.

The following guidelines will help you on your way (but remember, this is not the only way of programming and modeling this task!)

- Create another viewer class, this time with the purpose of rendering the buttons on the screen. As before the viewer class needs a reference to the box object. Create the different buttons (start with one!). For each button, add a method that adds a listener to the button.
- The control class should now also hold a reference to your "button viewer" class. Make inner classes that implements the ActionListener interface to perform different actions when the buttons are clicked (i.e. implement the method `actionPerformed`).
- The main method should now also have an object of your "button viewer" class.

Good to know:

`JPanel` is a class that can be extended and used to render buttons.

The `moveAllBalls` method will not work inside `actionPerformed` if not suitably modified. You need to start a new thread and make use of the `Runnable` interface.

You can use the `JComboBox` class to allow the user to choose which ball he or she wants to move. Take a look a the class in the API (it is not well documented in the book).