Design Pattern: Mediator

Martin Kjellin

May 10, 2014

The *mediator* design pattern [1, p. 273–282] decreases the coupling between interacting objects (which are commonly referred to as colleagues) by preventing them from referring to each other directly. Instead, the colleagues communicate with each other through a mediator object. The mediator's role is to route requests between the colleagues. This means that many-to-many interactions between colleagues can be replaced by one-to-many interactions between the mediator and the colleagues.

An object that has fewer connections to other objects is more likely to work correctly in isolation. This means that the mediator pattern makes code reuse easier. Also, since the mediator is responsible for the behaviour of the system of objects, developers can easily change this behaviour by altering only the mediator class.

A common example of the mediator design pattern is a dialog box with several widgets. In this case, each widget is a colleague. When a certain widget has changed (through user input), it notifies the mediator of the change (for example, by calling a method in the mediator). The mediator's task is then to update all other widgets that should be affected by the change. For example, the first widget might be a slider and the second might be a label displaying a value that is set by the slider.

The Solitaire application implements the mediator design pattern as part of the larger model–view–controller (MVC) pattern. An important aim of the MVC pattern is to decouple the model from the view, so that the state of the model can easily be displayed by different views. Furthermore, several objects in the application might need to be notified of a certain change in the model or the view, even if this is not the case in the application in its current state. Such notifications should preferably be delivered using a mechanism that relieves the developers of the burden of manually keeping track of all the recipients. This is why the mediator pattern is useful in the Solitaire application, as well as in others that feature a graphical user interface.

The mediator in the Solitaire application is an object of the gameStateController class that takes care of all communication between the model and the view. The communication is performed using PyQt's signals and slots mechanism, where each signal type is connected to one or several slots (methods). When a signal is emitted, the mechanism makes sure that the signal is received by the corresponding slot methods, which are then performed.

In some cases, the actions taken by the slot methods are very simple. For example, when the gameStateController object receives an updateStackSignal (which indicates that the content of the stacks on the game board has been changed) from the application's boardModel object, it just forwards the message

by emitting an updateSignal, which is received by the updateStacks method of the corresponding boardView object. The view is then updated according to the stack-content change.

In other cases, more complex procedures are necessary. For example, when a card is dropped on a stack, the corresponding stackView object emits a moveCardSignal. When this signal is received by the gameStateController object, it creates a moveCardCommand object and pushes this onto an undo stack. The pushing causes the redo method of the moveCardCommand object to be performed. In turn, this method calls the moveCard method of the boardModel, where the actual updating of the card position is performed. Finally, an updateStackSignal is emitted, and the procedure described in the previous paragraph is performed.

References

[1] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison–Wesley, 1995.