

# Chapter 29

## Model-View-Controller

### 29.1 The Problem

Most computer programs running on personal computers have a *graphical user interface*, involving several windows and input from both the keyboard as well as the mouse. A complex example of such an application is a *vector graphics editor* (see Figure 29.1) like for instance Adobe Illustrator, XFig, Inkscape, or Microsoft Visio. UML diagram editors can also be considered vector graphics editors. These present a palette of figures: rectangles, lines, ovals, etc., that the user can add to a drawing and next manipulate using the mouse. Several windows can be opened on the same drawing showing different parts and in different scales. In such a system the underlying drawing needs to be rendered in multiple windows at the same time while the drawing also needs to process input events from multiple windows. The challenge is to structure graphical applications such that coupling between the application's domain objects and the graphical views is low. Sidebar 29.1 is a war story of how early GUI builders often led developers to make a high coupling.

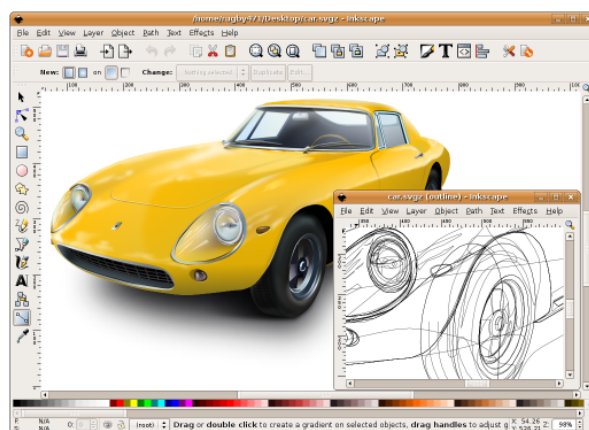


Figure 29.1: The Inkscape vector graphics editor.

### Sidebar 29.1: Early GUI Builders

I had the opportunity to use some early programs that allowed graphical user interfaces to be developed using a graphical editor. This was of course a big improvement to the former practice of writing GUI code by hand. I have wasted my fair share of hours making dialogs appear tidy by trial and error: *I try width 100 and height 20 pixels, (compile and run), oops, that was too much, let me try width 90, (compile and run), oops that was too little, ...*

However, all those builders that I have tried lured me into tight coupling between the view and the model code. Typically if I positioned a button on the dialog using my GUI builder I could click on the events associated with the button, like `mouseClick()`. If I selected that, an editor would pop up and I could enter source code directly to be executed when the button was clicked. The result was of course that model manipulation code was scattered all over the place in my GUI code. It made it difficult to overview and next to impossible to reuse if the same model operation was required by some other event, like a menu selection.

## 29.2 Model-View-Controller Pattern

The challenge facing a design to handle graphical user interfaces is actually twofold: first the need to keep multiple windows consistent and second to handle and interpret multiple input sources: keyboard, mouse, and maybe others. A well-established solution to this challenge is the MODEL-VIEW-CONTROLLER pattern, or just MVC pattern for short.

The first challenge is solved by the OBSERVER pattern. OBSERVER states that the underlying information, like the drawing's list of figures, must be stored in a **Subject** object that can notify its **Observers**, i.e. the windows rendering the drawing. In MVC these roles are called **Model** (containing state and notifying upon state changes) and **View** (rendering the graphics) respectively.

The second challenge is receiving and interpreting events from the user. To see why this is a problem consider what it means that a user clicks in a window. If the mouse hovers above a button it means that the button's command should be executed, if it hovers above a figure it means that the figure should become selected, if it hovers over some text in a text editor it means that the cursor position should be changed, etc. Thus, the same user event, a mouse click, has to be interpreted based upon the particular type of application and type of object.

👉 Consider some of the applications you use regularly (or start them) and think about the resulting behavior it exhibits when you click and drag with the mouse. Do you see different behaviors even in the same application depending upon its state?

The STATE pattern provides a compositional solution to this problem. Remember the intent of STATE. *Allow an object to alter its behavior when its internal state changes.*

Here it is obviously the window (**View** role) that receives input events but the resulting behavior depends upon the internal state: the type of application and the application's state. The **Context** role of STATE is the **View** role in MVC while the **State** role

is denoted **Controller**. However, the controller is more specific in that it has the responsibility to modify the model appropriately. The resulting structure contains the three parts: model, view and controller, as outlined in Figure 29.2, with the following responsibilities.

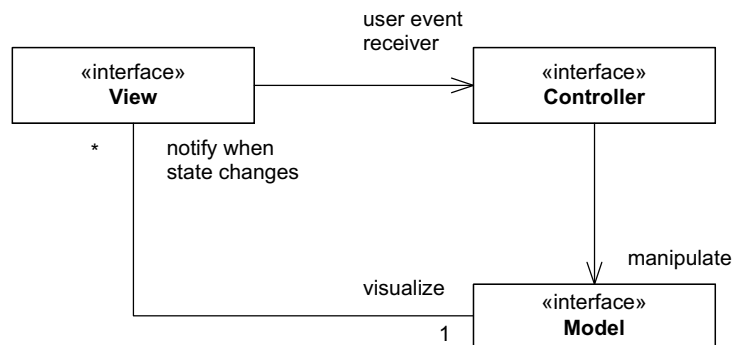


Figure 29.2: MVC role structure.

### Model


- Store application state.
- Maintain the set of Views associated.
- Notify all views in case of state changes.

### View

- Visualize model state graphically.
- Accept user input events, delegate them to the associated Controller.
- Manage a set of controllers and allow the user to set which controller is active.

### Controller

- Interpret user input events and translate them into state changes in the Model.

 As the controller usually translates simple input events into the relevant method call on the model (for example mouse drag events into `move()` calls on a figure) the controller actually acts as an ADAPTER for the model.

The protocol of MVC is shown in Figure 29.3. Only the notification protocol is shown. Of course views must also register in the model to receive update events, and the view must be associated with the proper controllers.

The MVC pattern is the fundamental pattern in the MiniDraw framework, described in learning iteration 7.

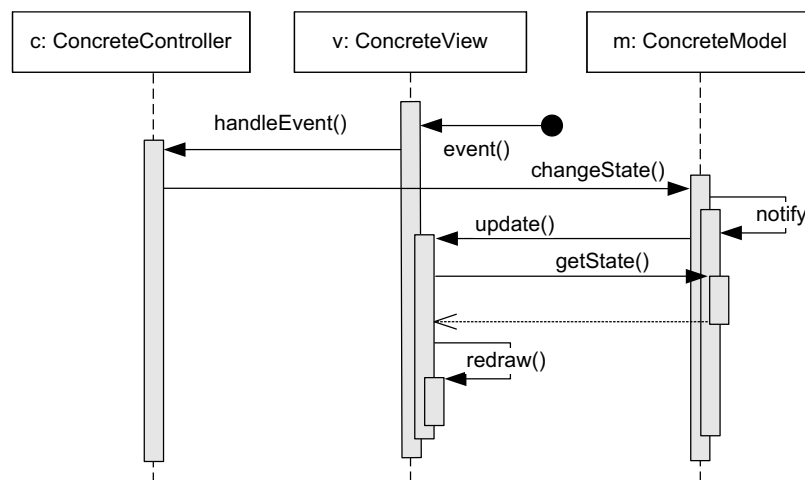


Figure 29.3: MVC protocol.

## 29.3 Analysis

The MVC patterns of course inherits the benefits and liabilities of OBSERVER and STATE. First, the broadcast mechanism of OBSERVER ensures that the user can open as many windows as he or she likes and they are all ensured to be consistent and to show the same model—any change in the model are reflected in all windows. The coupling is loose so any window may choose to render the model as it sees fit, as exemplified in Figure 29.1 where one window shows a wireframe image of the car model while the other shows a fully rendered car image. The user event delegation to the **Controller** objects ensures that model changes are not tightly coupled to the particular window object and that they can be changed at run-time. Consider the palette in a graphics editor: choosing the rectangle tool over the oval tool changes what is drawn with the mouse. This simply maps to changing the controller associated with the window which demonstrates how easy it is to change behavior at run-time.

The liabilities of the pattern are the classic ones. The structure and protocol is a bit complex and care must be exercised during programming to get it right. The problems of multiple updates and circular update sequences are inherited from the OBSERVER pattern and have to be considered.

MVC is an **architectural pattern**. Design patterns are not tied to any particular domain but can find usage in more or less all types of applications. Architectural patterns are more coarse grained and focus on a particular architectural problem and a particular domain. MVC presents a qualified solution to the problem of structuring graphical user interfaces and of course if your application does not have such it is of no interest. MVC is also not the only possible solution—others exist such as the PRESENTATION-ABSTRACTION-CONTROL pattern (Buschmann et al. 1996).

In the presentation so far, the **Model** is shown as a single object. In practice, a model is often complex and consists of a lot of objects with different interfaces. It is up to the concrete design decision whether the controller should access individual objects in the model or if it is beneficial indeed to have a single interface for manipulating all aspects of the model. In the latter case, the model interface becomes a FACADE.

The MVC pattern is treated in detail by Buschmann et al. (1996). The pattern was first described by Reenskaug (1978).

## 29.4 Review Questions

Describe the type of applications that MVC outlines a plausible architecture for.

Name the three basic roles of the pattern and describe the responsibilities of each. How are the three roles related (type of relation and multiplicity)? What is the protocol once an application using the pattern is running?

Describe benefits and liabilities of the pattern.

Describe why MVC is an architectural pattern rather than a design pattern.

## 29.5 Further Exercises

**Exercise 29.1.** Source code directory:

`chapter/facade`

Review the GUI for the pay station from Chapter 19, *Facade*. Analyze the production code and evaluate whether the MVC pattern has been used to structure the GUI.

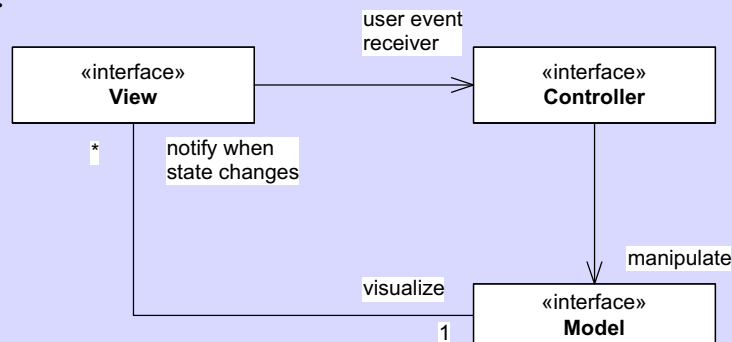
## [29.1] Design Pattern: Model-View-Controller

**Intent** Define a loosely coupled design to form the architecture of graphical user interfaces having multiple windows and handling user input from mouse, keyboard, or other input sources.

**Problem** A graphical user interface must support multiple windows rendering different visual representations of an underlying set of state-full objects in a consistent way. The user must be able to manipulate the objects' state using mouse and keyboard.

**Solution** A **Model** contains the application's state and notifies all **Views** when state changes happen. The **Views** are responsible for rendering the model when notified. User input events are received by a **View** but forwarded to its associated **Controller**. The **Controller** interprets events and makes the appropriate calls to the **Model**.

**Structure:**



**Roles** **Model** maintains application state and updates all associated **Views**. **View** renders the model graphically and delegates user events to the **Controller** that in turn is responsible for modifying the model.

**Cost - Benefit** The benefits are: *loose coupling between all three roles* meaning you can add new graphical renderings or user event processing. *Multiple views/windows* are supported. It is possible to *change event processing at run-time*. The liabilities are: *unexpected/multiple updates*: as the coupling is low it is difficult for views to infer the nature of model state changes which may lead to graphical flickering. *Design complexity* is another concern if the development team is untrained.