The Model-View-Controller Design Pattern

By Maria Litvin and Gary Litvin

Introduction

In OOP, much of the emphasis shifts from software development to software design. Object-oriented design (OOD) is not easy — designing a software application often takes more time than coding it, and design errors usually take longer to correct than errors in the code. *Design patterns* represent a bold attempt by experienced software designers to formalize their skill and share it with novices. Of course, each software application is unique and its design is unique, but familiarity with design patterns can help you solve common design problems and avoid common mistakes.

No one has come up with a good formal definition of a "pattern" — but somehow, most people recognize a pattern when they see one. In fact, we humans are very good at pattern recognition. We recognize as a pattern any structure, idea, or behavior that recurs in diverse situations. We talk about organizational patterns and patterns of behavior, grammatical patterns, musical patterns, speech patterns, and ornament patterns. Recognizing patterns helps us structure our thoughts about a situation and draw on past experiences of similar situations. The word *pattern* therefore has a second meaning as well: an example worthy of imitation.

The idea of design patterns in OOP can be traced to the influential writer on architecture, Christopher Alexander [1]. In his books, *The Timeless Way of Building* [2] and *A Pattern Language* [3], Alexander introduced design patterns as a way to bring some order into the chaotic universe of arbitrary architectural design decisions: how rooms should be connected, where windows should be placed, etc. In *A Pattern Language*, Alexander and his co-authors catalogued 253 patterns that helped solve specific architectural problems and offered standard ideas for better designs.

The first and most famous book on the subject of software design patterns, *Design Patterns*, by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, was published in 1995 [4]. It is known as the "gang of four" book. Since then, hundreds of patterns have been published, some rather general, others specialized for particular types of applications. Both academia and the software industry have taken a great interest in design patterns. This interest is evident from the numerous conferences, workshops, discussion groups, and web sites dedicated to collecting and cataloguing patterns [5, 6].

A more or less standard format for describing patterns has emerged. A typical description includes the pattern name; a brief statement of its intent or the problem it solves; a brief description of the pattern and the types of classes and objects involved; perhaps a structural diagram; and an example, sometimes with sample code. Such descriptions are often quite technical and terse. In this paper we present a less formal description of MVC along with a complete program example that illustrates its use.

The Main Principles of OOD

Object-oriented design has two main objectives: to make the classes *cohesive* and to reduce *coupling*. Cohesion means that each class of objects defines a small set of behaviors and responsibilities unified by a common purpose. Apples belong with apples, and oranges with oranges — a fruit basket of a class is not a good idea. Coupling refers to the degree of interdependency between classes: the less each class needs to "know" about the other classes, the better.

These two objectives are somewhat in tension. To accomplish anything useful, cohesive classes need to work together. An OO designer is thus constantly striving to find a balance, to come up with a set of cohesive classes with relatively low coupling but capable of working together.

One corollary of the cohesion principle is the separation of the user interface from the computational model. This is desirable for many reasons. In particular:

- In defining how an abstract model should work, it is easier simply to assume that it gets some inputs and gives some outputs without considering precisely how this happens;
- In many programs, the user interface will need to be updated or customized more often than the underlying model;
- Developing GUI (graphical user interface) requires a different set of programming skills than those needed for developing data structures and algorithms.

Separating GUI from the computational model poses a number of questions, however. How does GUI interact with the model? Who is in charge? The Model-View-Controller (or MVC for short) design pattern helps address these questions.

MVC

MVC typically applies to software applications in which the user interacts with the program in real time. Every software application models some situation or process. In OOP, a model is described by a class or a system of related classes. For example, a chess game can be modeled by a list of moves; a moving car can be modeled by a set of variables (position, velocity, acceleration) and the equations that connect them; a document in a word processor can be modeled by a list of its components, including text boxes, pictures, and embedded formatting tags. According to our separation principle, the model should be abstract, divorced from any GUI.

The state of the model is described by the current values of its internal variables (data fields). Suppose the model provides modifier methods that can change those values, so the state of the model can change in response to "outside" stimuli. In the MVC design, objects that cause the model to change are called *controllers*. For example, in a computer chess program, one object may represent a human player and another a computer player. Both act as controllers: either can change the model (the chess game) by making a move. A moving car can be controlled by an object that represents the gas pedal and also by a clock that periodically moves the car. A document in a word processor can be changed by the buttons on the toolbars and menus and by

input from the keyboard and mouse. In a GUI application, the controllers can act independently of each other and do not have to be synchronized in any way.

All of the above is quite straightforward and familiar. Interesting things begin to happen when you need to monitor the changes in the model, especially if you need to have several different "views" of the model. In a chess game, for example, you might want to see the current board position and/or a list of the moves that led to it; for a moving car, you might want to see a picture of a car moving on the road or the car's speedometer gauge, or both at once; in a word processor you might want to switch between "normal," "page layout," and "outline" views.

The central question that MVC addresses is: How do the views get updated? You might be tempted to set up a "totalitarian system" in which one central controller updates the model and manages all the views (Figure 1). In this approach, the model is not aware that it is being watched and by whom. The controller changes the model by calling its modifier methods, gets information about its state by calling its accessor methods, and then passes this information to all the views.

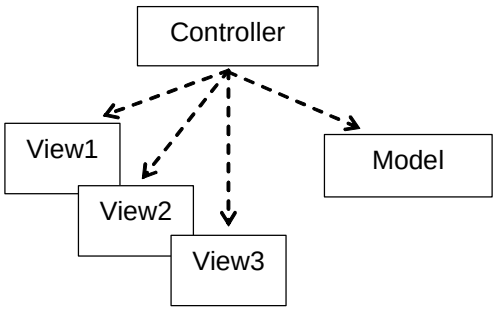

Figure 1. "Totalitarian" control of the model and the views

This setup becomes problematic if requests to change the model come from many different sources (GUI components, keyboard and mouse event listeners, timers, etc.), so that it is not practical to combine all of them in one class. All the different requests would have to go through the cumbersome central bureaucracy. If you split the control mechanism into several classes, you will end up with an "oligarchy," in which every controller updates every view (Figure 2). Forget about low coupling.

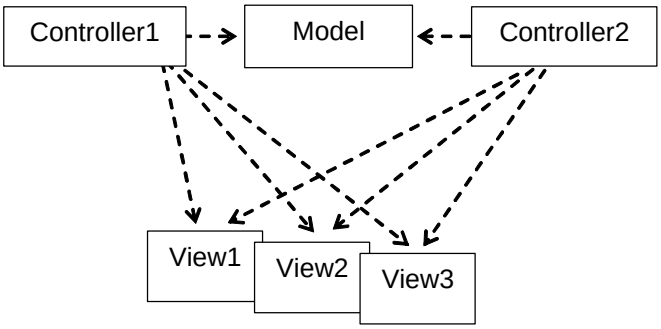

Figure 2. An "oligarchy": each controller can update the model and all the views

An MVC design offers a decentralized solution in which the views are attached to the model itself. The model knows when its state changes and updates the views when necessary (Figure 3). Such design can support several independent views and controllers and makes it easy to add more views and controllers.

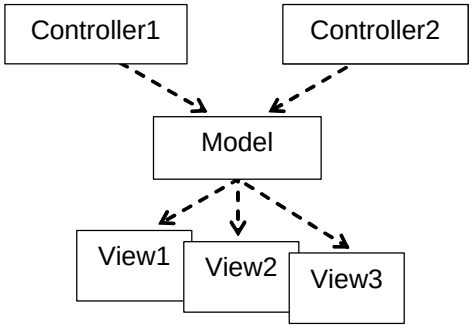

Figure 3. MVC design: different views are attached to the model

The term "view" implies a visible GUI component that provides a visual representation of the model, but it doesn't have to be. Any object that monitors the changes in the model — a file that keeps a log of the changes, a sound player that plays a particular audio clip at the appropriate moment, even some other model — can qualify as a "view."

Observer-Observable

The Java library supports the "MV" part of the MVC design pattern through the `Observable` class and the `Observer` interface. Both belong to the `java.util` package. (We are used to interface names that sound like adjectives that end with "-able," and class names that sound like nouns. Here it is the reverse.) `Observable` and `Observer` work together. The class that represents the model extends `Observable`; the class that represents a view implements

`Observer`. Thus the model becomes `Observable` and a view becomes an `Observer`. An `Observable` object maintains a list of its observers.

Let's look at the `Observer` interface first. It specifies only one method:

```
void update(Observable model, object arg);
```

`update` is called for each observer in the model's list of observers when the model wants to update its observers. (This mechanism is hidden in `Observable`'s `notifyObservers` method). `update` receives a reference to the model as the first parameter. `update`'s code often casts that parameter back into the specific model type to be able to call its accessor methods. For example:

```
import java.util.Observable;
import java.util.Observer;
...

// Represents a roadside display of a motorist's speed.
public class RoadsideDisplay extends JPanel
    implements Observer
{
  private JLabel speedDisplay;

  public RoadsideDisplay()
  {
    // Set up GUI:
    ...
  }

  public void update(Observable model, Object arg)
  {
    Car car = (Car)model;
    double speed = car.getSpeed();
    speedDisplay.setText(String.format("%5d  ", speed));
  }
}
```

`update`'s second parameter (argument) `arg` is an optional parameter that can be passed to `update` by the model to convey some additional information. If the model does not provide it, then it is set to `null`.

Note that when a view is a *Swing* component that overrides the `paintComponent` method, `update` cannot call `paintComponent` directly. It has to save a reference to the model in a private field, then call `repaint`. For example:

```
import java.util.Observer;
import java.util.Observable;
...

// This class represents a speedometer gauge on the screen.
public class Speedometer extends JPanel
    implements Observer
{
  private Car car;

  ...

  public void update(Observable model, Object arg)
  {
    car = (Car)model;
    repaint();
  }

  public void paintComponent(Graphics g)
  {
    super.paintComponent(g);

    if (car == null)
      return;  // to avoid the NullPointerException when paintComponent
               // is called before update
    double speed = car.getSpeed();
    ... etc.
  }
  ...
}
```

Now let's take a look at `Observable`. An `Observable` object (the model) maintains a list of all observers attached to it and provides methods for adding an observer and for notifying all its observers that they need to be updated. A programmer typically uses the following three methods of the `Observable` class:

```
public void addObserver(Observer view)  // Adds view to the list of observers

public void setChanged()                // Sets the changed flag to true

public void notifyObservers()           // or notifyObservers(arg);
                                        // If the changed flag is set, calls
                                        //   update for each observer
```

`addObserver` can be called from `main` or another method that creates the observer. For example:

```
Car car = new Car();
...
RoadsideDisplay roadsideDisplay = new RoadsideDisplay();
car.addObserver(roadsideDisplay);
Speedometer speedometer = new Speedometer();
car.addObserver(speedometer);
...
```

The `setChanged` method is called by the model itself; it simply sets the `changed` flag to `true`. `setChanged` is usually followed by a call to `notifyObservers`. For example:

```
// Represents a model for a vehicle moving along a straight line.
public class Car extends java.util.Observable
{
  ...

  // Moves this car for the time increment dt.
  public void move(double dt)
  {
    ...
    position += speed * dt;
    ...

    setChanged();
    notifyObservers();
  }
}
```

`notifyObservers` checks whether the `changed` flag is set, and, if so, calls the `update` method for each observer in the observers list. It then resets the `changed` flag to `false`. `notifyObservers()`, the no-args version, passes `this` to the `update` method as the first parameter and `null` as the second parameter. The overloaded version of `notifyObservers` that takes one parameter, `Object arg`, passes `arg` as the second parameter to the `update` method. The Observer-Observable mechanism has one limitation: it does not allow the model to update its observers selectively — it's either all or none. The model can use the `arg` parameter (for example, an `enum` value or a string) to instruct some of the observers to take a specific action or to ignore the `update` call altogether.

MVC in Action: An Example

Our example is indeed a toy car ride simulator. We chose this example for several reasons:

- The situation is familiar to everyone;
- The model is simple but not trivial;
- The allegory of views and controllers is obvious;
- The situation naturally calls for several different kinds of controllers and views;
- All the classes are short and cohesive;
- The MVC design is an obvious choice, and in fact it would be hard to design this application without it.

Figure 4 shows a snapshot of the program screen. It shows five windows: a speedometer gauge, a gas pedal, a brake pedal, a roadside speed display, and a picture of a road with a car on it. When the user presses the gas or brake pedal, the car accelerates or slows down accordingly, and its speed is displayed on the speedometer and the roadside display. We have put each component into a separate window to underscore their independence from each other.

In this example we have a model, three controllers, and four views. The model represents the position, speed, and acceleration of a car moving along a straight line, with the given levels of gas or brakes applied. The gas and brake "pedals" serve as controllers. The third controller is invisible: it is the timer that moves the car, creating the animation effect. The three views visible

in Figure 4 are the speedometer gauge, the roadside speed display, and the "road" with the moving car picture on it.

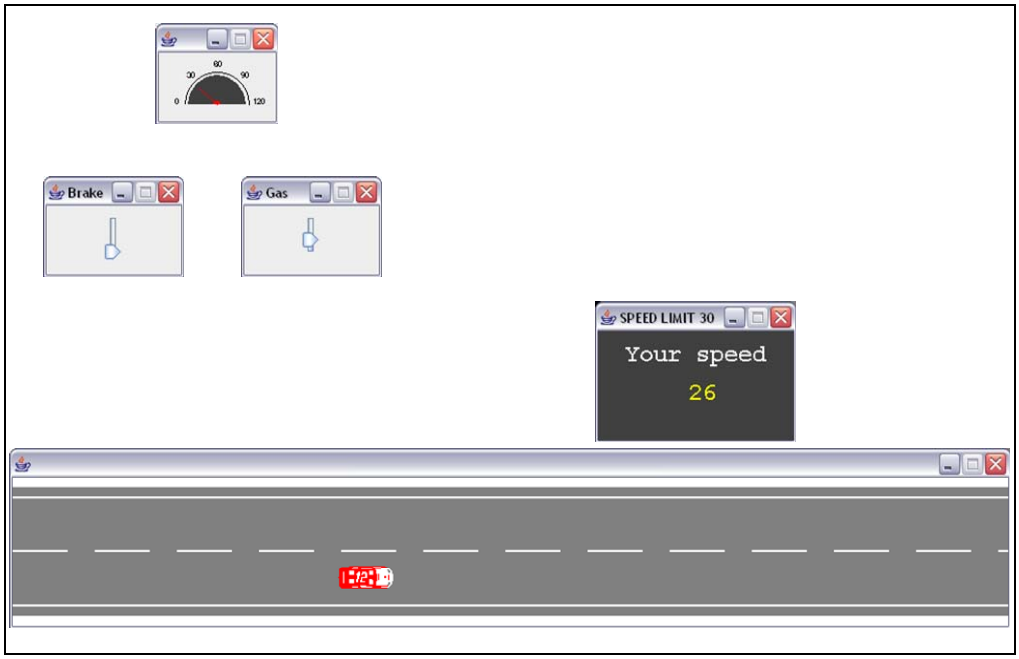

Figure 4. The windows from the *MVCDemo* program

We mentioned earlier that a "view" does not have to be visible. To demonstrate this (and just for fun) we have added a fourth "view" class to this program: `TiresSqueal`. It plays a sound like squealing tires when the car brakes too fast. Like the other view classes, `TiresSqueal` implements `Observer`.

A `Car` object has the modifier methods `applyGas`, `applyBrakes` and `move`, which are called by the respective controllers. It is the model that decides when to notify the views about a change. Here only the car's `move` method notifies the views; the `applyGas` and `applyBrakes` methods do not.

Figure 5 shows a class diagram for this program. The class `Car`, the model, extends `Observable`. The view classes, `Speedometer`, `RoadsideDisplay`, `Road`, and `TiresSqueal`, implement `Observer`. (Library classes and interfaces are usually not included in such diagrams, but we have included `Observable` and `Observer` to show where they fit in.) The `BrakePedal` and `GasPedal` classes are derived from an abstract class `Pedal`, derived from `JSlider`, modified to release the pedal when the mouse button is released. The `Clock` class has a timer that fires every 50 milliseconds; it calls the car's `move` method each time it fires. `MVCDemo`'s `main` method creates the model and all the views and controllers.

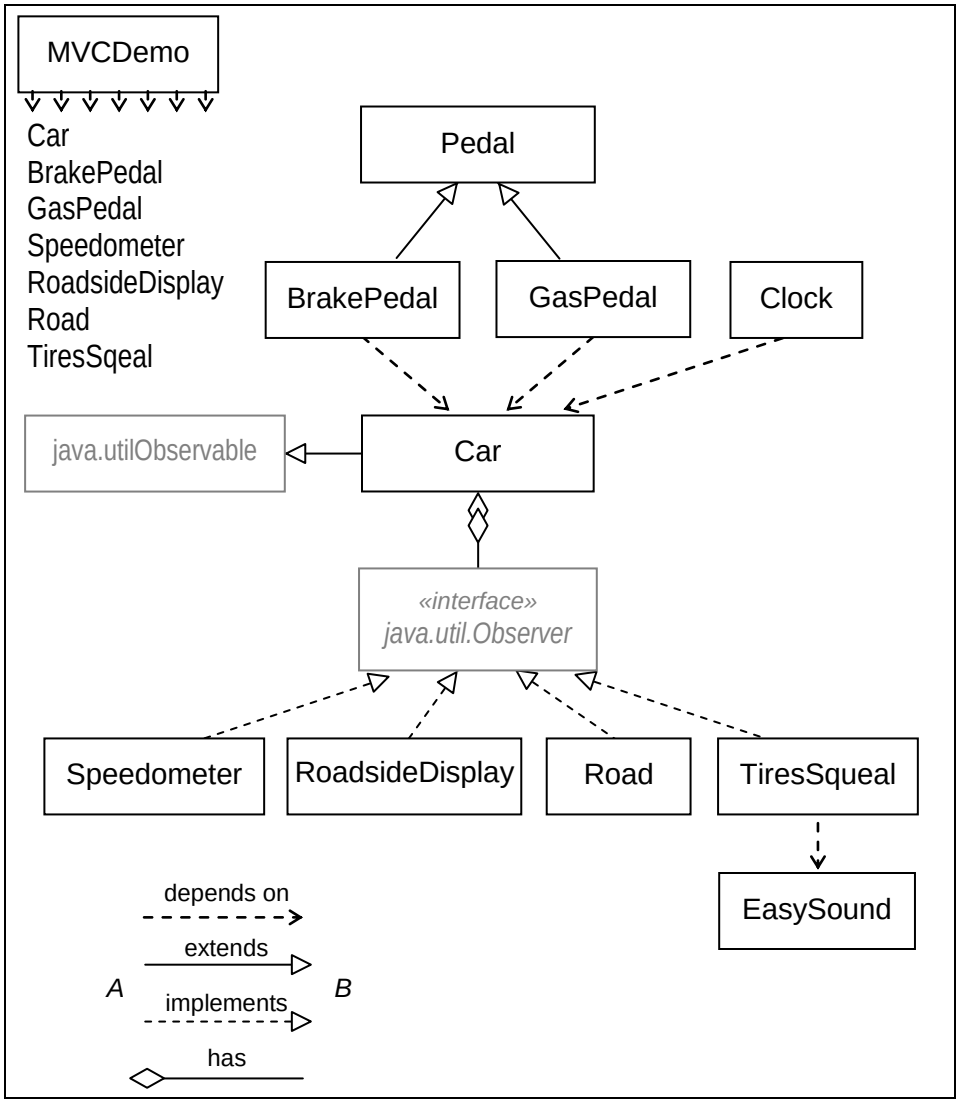

Figure 5. A class diagram for the *MVCDemo* program

In this design, the classes know very little about each other. Even the `Car` class is not aware of the particular types of observers attached to it: `Car` interacts with its observers via the `Observer` interface. A diagram like this, without lines crisscrossing each other, indicates that the coupling in our design is minimal. The `MVCDemo` class is aware, of course, of all the classes in the project, (except `EasySound`), because its `main` method creates the model and all the views and controllers. The classes are cohesive, with well-defined limited responsibilities. We could have put the timer inside the `Car` class, for example, but that would violate the cohesiveness principle.

Note how flexible this design is. To integrate `TiresSqueal` into the program, for example, all we had to do was write that class and add one line to `MVCDemo`'s `main`:

```
car.addObserver(new TiresSqueal());
```

All the other classes remained intact. (`TiresSqueal` uses our `EasySound` class to load the sound clip and play the sound.)

The complete source code for this program and a runnable `jar` file are available at http://www.skylit.com/oop/index.html.

References:

[1] *Christopher Alexander: An Introduction for Object-Oriented Designers* by Doug Lea, http://g.oswego.edu/dl/ca/.

[2] C. Alexander, *The Timeless Way of Building*. Oxford University Press, 1979.

[3] C. Alexander, S. Ishikawa, and M. Silverstein, *A Pattern Language*. Oxford University Press, 1977.

[4] *Design Patterns: Elements of Reusable Object-Oriented Software*, by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Addison-Wesley, 1995.

[5] http://hillside.net/patterns/

[6] http://www.exciton.cs.rice.edu/JavaResources/DesignPatterns/