Refactoring Model-View-Controller[*]

Stuart Hansen
Department of Computer Science
University of Wisconsin - Parkside
Kenosha, WI 53144
(262) 595 - 3395
hansen@cs.uwp.edu

Timothy V. Fossum
Department of Computer Science
University of Wisconsin - Parkside
Kenosha, WI 53144
(262) 595 - 2297
fossum@cs.uwp.edu

ABSTRACT

Model-view-controller (MVC) is an important architectural design pattern that frequently does not receive the attention it deserves. It is generally treated as a prescriptive design pattern, where students are taught to use three different categories of objects to construct GUI programs. There are subtle and important aspects of MVC that anyone developing GUIs should know. We approach these issues in our classes by taking relatively simple GUI programs and refactoring them. Our emphasis is on examining the strengths and tradeoffs involved with the various design decisions. We illustrate our approach in this paper using a simulated stopwatch.

1. INTRODUCTION

Developing graphical user interfaces has been written about extensively [1, 2, 5, 6, 7, 9, 10]. There are numerous libraries and toolkits for developing GUIs. Textbooks frequently take the approach of demonstrating how to use a particular library or toolkit, spending little time discussing the structure of the program or architectural patterns like MVC. As computer science educators, it is important that we go beyond studying a library or toolkit and emphasize the design decisions that guide the development of an application. When introducing GUI programming in our classes, we use an iterative approach, refactoring our designs and implementations, to offer students a deeper understanding of the features and tradeoffs of various design decisions.

2. MODEL-VIEW-CONTROLLER

Model-view-controller is the catch-all phrase used to describe the design concerns of developing GUI applications in an object-oriented language. MVC originated with Smalltalk-80, but has been used to describe approaches for developing GUI programs in different languages [3, 4, 8]. The examples presented in this paper

[*] Supported in part by NSF grant DUE #0089406

were implemented using Java Swing, but the same designs may be used to implement the examples in Visual Basic, C#, or any other language supporting GUIs.

MVC is frequently presented as requiring the programmer to use three types of objects to develop their systems: models, views and controllers. Models contain the data of the system, as well as methods to access and manipulate the data. Views present the data to the user. Controllers process user input and update the model and view appropriately. This approach nicely separates three basic concerns of the system, thus promoting code maintenance and reuse. Each application, however, has subtle issues that must be thought through and addressed. The central problem is that the inherent coupling among the model, the views and the controllers. The way that coupling is realized in an implementation has a profound impact on how easy the code is to develop, maintain and reuse.

2.1 The Stopwatch Example

We illustrate our ideas in this paper with different implementations of a stopwatch simulator. Stopwatches are used to time athletes and other activities. One standard model of a stopwatch has a digital display of the elapsed time and two buttons to control its operation. A simple interface is shown in Figure 1.

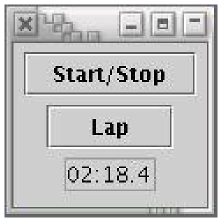

Figure 1. A simple interface for a two button stopwatch.

The display shows the elapsed time since the watch was started. The Start/Stop button starts and stops the watch.

The Lap button has multiple responsibilities. It allows the user to measure 'split' times without completely stopping the watch. For example, in a relay race multiple athletes compete sequentially as a team. The first runner (or swimmer, or skier, or ...) finishes their portion of the race before the second one starts. Coaches are interested in the overall performance of the relay team, but they are also interested in the performance of each athlete. They measure individual performances using the Lap button. Clicking the Lap button freezes the display, but the stopwatch keeps running in the background. Clicking the Lap button a second time unfreezes the display and updates the displayed time to the current elapsed time. If the watch is stopped, clicking the Lap button resets the time to 0. If the watch is stopped while the display is frozen, clicking the Lap button updates the display to the final elapsed time. A second click is necessary to reset the watch.

The stopwatch is a simple, but fairly typical GUI program. It is characterized by being in a stable state that changes only when events occur. It uses external events, e.g. button clicks, for user interaction. It also uses internal events, e.g. timer firings, to update the elapsed time.

2.2 The Two Button Stopwatch Design

Our first stopwatch design uses a typical approach to MVC. The application inherits from the window class (`JFrame` in Swing). The GUI components are added to the window. The model consists of variables and methods within the class. Control consists of inner classes and methods that process events.

2.2.1 The View

The stopwatch view is shown in Figure 1. It consists of a window with two buttons and a textbox.

2.2.2 The Model

The model consists of two longs, each representing time, and their associated methods.

`elapsedTime` contains the total time that has elapsed while the stopwatch is running.

`displayTime` contains the time currently displayed by the stopwatch.

The model also contains methods to set and get each of the values. `setDisplayTime(long)` not only sets `displayTime`, but translates the new value to a `String` and updates the view using the `String`.

2.2.3 The Stopwatch's Control

Understanding the control of the stopwatch takes a bit of work. Too often control is defined as the event handlers placed behind the input widgets. Control more appropriately refers to the code that directs the overall actions of the system. Frequently this involves more classes and objects than just the event handlers. As we shall soon see, in our approach the event handlers regularly delegate their responsibilities to other control objects.

As with many GUI applications, the stopwatch's control is state based. Figure 2 shows the state diagram for the stopwatch. The circles represent the control states. The arrows between the states are transitions triggered by events. The stopwatch is in one of four different control states. The top two states are `Stopped` and `Started`. The stopwatch is initialized in the `Stopped` state. Clicking the StartStop button carries the watch to the `Started` state. Clicking it again stops the watch. The bottom two states occur after the Lap button has been pressed. The display is frozen in these states. They are distinguished from each other because the watch may or may not be running in the background.

Each arrow in the state diagram represents either a button click or a timer event. The arrows are labeled with the event, followed by any actions that take place when the event occurs. For example, pressing the Lap button freezes and unfreezes the display. Unfreezing occurs with both upward pointing arrows in the diagram. When unfreezing, the system should also immediately update the `displayTime` to the current `elapsedTime`. This is abbreviated in the diagram as "update display." Similarly, if the watch is stopped and the Lap button is clicked, both the elapsed time and the display time are reset. This event appears as the semicircular arrow in the upper left of the diagram.

The diagram also shows the internal timer events. When the watch is started a timer begins firing every 0.1 seconds. This event updates the `elapsedTime` and, if not frozen, the `displayTime`. The timer events are the two semicircular arrows in the upper and lower right of the figure.

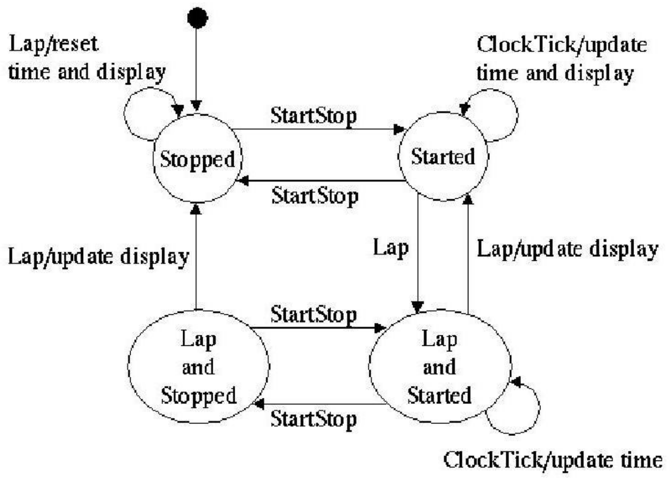

Figure 2. The state diagram for the two button stopwatch. There are four control states, represented by the circles. The transitions between them are represented by the arrows. The arrows are labeled with the event that causes the transition and any actions that occur when the event fires.

2.2.4 Implementing State Based Control

The State Design Pattern is an elegant approach for implementing state based control [8]. An interface named `State` is declared with public methods for each of the possible events. There are three events in our state diagram. The corresponding methods are: `startStopPushed()`, `lapPushed()`, and `updateTime()`. A separate class implementing `State` is written for each control state. The methods of each of these classes contain the code that implements the state transitions and associated actions for each of the events. A variable of type `State` is added to the application and initialized to the state diagram's start state, in our case `Stopped`.

For example, consider what happens when the Lap button is clicked in the `LapAndStarted` state. This is the upward pointing arrow on the far right of Figure 2. Two things happen, the display time gets updated and the control state transitions to `Started`. The `LapAndStarted` class contains the following method to accomplish this.

```
public void lapPushed()  {
   setDisplayTime(getElapsedTime());
   setState(new Started());
}
```

The interfaces and classes involved in implementing the state diagram are shown in Figure 3.

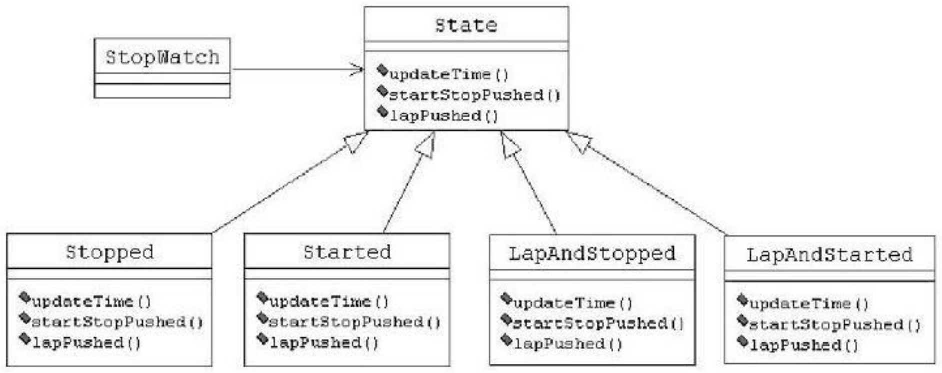

Figure 3. The classes and interfaces involved in implementing the state diagram in Figure 2. Each of the states in Figure 2 becomes a class. Each of the events in Figure 2 becomes a method in each of the classes.

The state design pattern simplifies the development and maintenance of state machine code. There are no if statements or bit fields to check to determine the control state of the system. Instead, polymorphism determines the state at runtime using the `state` variable. When `state` refers to an instance of `Started`, the actions carried out are those associated with `Started`. When `state` refers to an instance of `LapAndStarted`, the actions carried out are those associated with that state.

3. REFACTORING THE STOPWATCH

The two button stopwatch simulation is similar to many real stopwatches and the design just presented, while quite simple, illustrates a standard approach to MVC. The model, view and control components are all present and relatively easy to understand. It has a major flaw, however, in that it is not very extensible. For example, it would be nice to extend our system to time multiple athletes simultaneously. During the recent Olympics, swimmers swam in an eight lane pool. All of them were timed by a single electronic clock. Each lane, however, received its own lap times as that swimmer finished a lap. This type of extension is very difficult to achieve with our current design, as there is only one `displayTime` variable, one textbox to display its value, and one Lap button to freeze/unfreeze it. Adding the additional code to handle this extension greatly increases the complexity of the model, the view and the controllers.

3.1 Decoupling Model and View

Models and views are intrinsically coupled. A view represents a model to the user. Our two button stopwatch recognizes this and embraces it by placing both the model elements and GUI widgets in the application class. This keeps them visible to each other and makes updating the view based on model data a bit easier.

A better approach, however, is to accept that there always will be coupling between the model and view, but try to decouple them as much as possible in the source code. There are two very good reasons to do this. The first is to make mixing and matching models and views easier. It is not uncommon for modern applications to have different views of the same model. In our Olympic swimming example, we want to have multiple views of the stopwatch, one for each lane of the pool. The other reason to decouple model and view is for development and maintenance. If model and view are decoupled, different programmers can be responsible for each.

In an object-oriented world, decoupling model and view means using different classes and objects for each. A key realization for good decoupling is that both model and view contain data. We typically think about data as belonging solely to the model, e.g. `elapsedTime`, but the view also contains data. In our example, `displayTime` belongs to the view. Making `displayTime` part of the view facilitates multiple views, each potentially displaying a different time.

Unfortunately, decoupling models and views requires duplicate information to be stored in each. When the stopwatch is running, `elapsedTime` and `displayTime` contain the same information and need to remain coherent. Changes in the model must be reflected in the view as soon as possible. The more decoupled the model and view become, the harder the programmer has to work to make certain `displayTime` has the proper value. Event driven programming supplies an ideal basis for implementing the decoupling, while solving the coherence problem. Changes to the model fire events to all registered views. The event handlers are responsible for making the appropriate updates to the views. The Java class libraries even contain the `PropertyChangedEvent` class specifically designed for this purpose.

3.2 Refactoring Control

The other major challenge in separating the model from the view is decoupling the control of each. The events in Figure 2 have actions that influence both the model and the view. For instance, clicking the Lap button generally only freezes and unfreezes the display, but if the stopwatch is stopped, it resets both `elapsedTime`, which is part of the model and `displayTime`, which is part of the view. Decoupling model and view requires us to separate the model events from the view events.

3.2.1 *The Three Button Stopwatch*

The main problem is that the Lap button affects the model in the `Stopped` state, but not elsewhere. We would like the Lap button to affect just the view. Ideally, the Lap button should toggle between `Stopped` and `LapAndStopped`, just as it already toggles between `Started` and `LapAndStarted`. We introduce the desired symmetry by adding a Reset button to the stopwatch. Figure 4 shows the three button stopwatch interface.

The Reset button replaces the Lap button in the `Stopped` state. The Reset button is only active when the stopwatch is stopped. If pressed in this state, the watch is reset. In all other states the button does nothing. It is worth noting, too, that real stopwatches are often designed with three buttons that function just as we have described. To implement the reset button, `resetPressed()` is declared in the `State` interface and implemented in each of the state classes. In all state classes, except for `Stopped`, it is a method with an empty body.

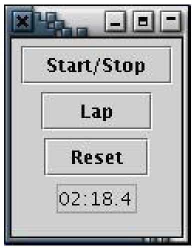

Figure 4. The interface for a three button stopwatch.

3.2.2 *Separating Model and View States*

Figure 5 shows the state diagram for the three button stopwatch. The arrow in the upper left now represents the new Reset event. The semantics of the Lap button are redefined so that it toggles between `Stopped` and `LapAndStopped`. The events in the state machine are now symmetric, with the left and right halves very similar and the top and bottom halves also similar.

Recall that the main reason for introducing the Reset event is to facilitate refactoring the state diagram into model control and view control. This is relatively easy to do starting from the state diagram in Figure 5. Each horizontal pair of states represents control of the model. That is, the model is either Stopped or Started. Each vertical pair of states represents control of the view. The view is either Frozen or Updating. We refactor the state machine into two separate state machines, one for controlling the model and the other for controlling the view.

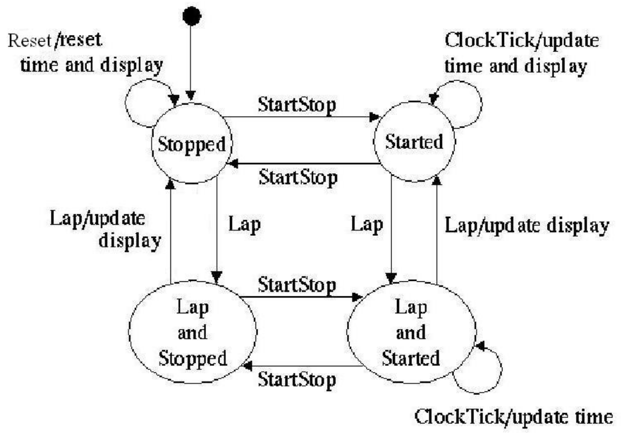

Figure 5. The state diagram for the three button stopwatch.

Figure 6 shows the state diagram for controlling the model. The Reset button, the StartStop button and the timer each generate events that are handled only by this state machine.

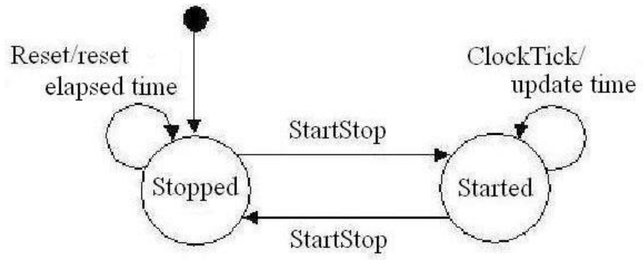

Figure 6. The redesigned state diagram for controlling the model.

3.2.3 The Coherence Problem Revisited

Figure 7 shows the state diagram for the view states. A stopwatch view is either Updating or Frozen. If there are multiple views, we duplicate this state machine in each. This lets each view be Updating or Frozen independently. There are several new features in this state diagram. The *Time changed* event is new. It is an internal event that is fired by the model to the views whenever `elapsedTime` changes. This event propagates the new `elapsedTime` to each display. The views receive the event and update `displayTime` when they are in the `Updating` state.

Another new feature in this diagram is the *update display* action on the Lap event when unfreezing the view. For this action, the view must be able to query the model to obtain the current `elapsedTime`. This is because the stopwatch may be stopped and therefore the Time changed events won't be occurring. Instead of being just a passive recipient of events, the view must actively request `elapsedTime` from the model.

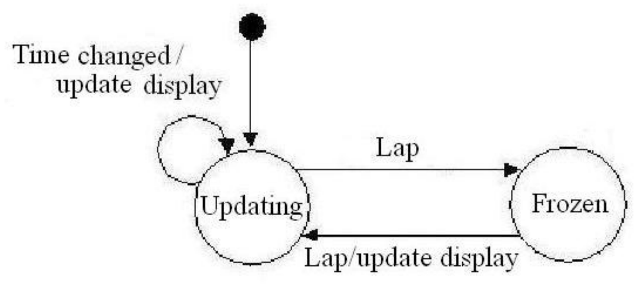

Figure 7. The state diagram for views.

3.3 Implementing Multiple Views

Separating model and view means multiple views can now be associated with the same model. The model's control unit gets its own window. Figure 8 shows this window for the stopwatch. The StartStop and Reset buttons belong to this control unit. The events from this control unit propagate themselves to the model and are handled as defined in the state diagram shown in Figure 6.

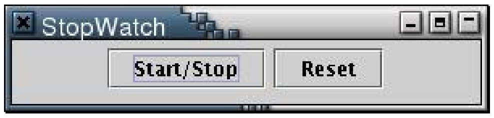

Figure 8. The control unit for the stopwatch.

Figures 9 and 10 show two possible views, one analog and one digital, that can concurrently be associated with the model. A view has a GUI representation of its `displayTime` and a Lap button. The Lap button only influences its view. One view may be frozen while the others continue updating.

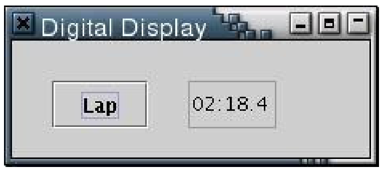

Figure 9. The digital view of the stopwatch

It is also possible to have multiple instances of the same view associated with a model. This brings us back to our Olympic swimming example. We can associate eight views, one per lane, with the stopwatch. Each view records the lap times and final time for its lane.

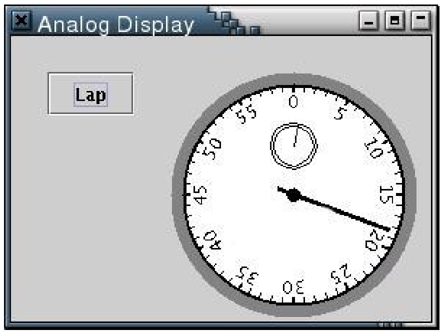

Figure 10. The analog view of the stopwatch.

4. CONCLUSION

We have used our stopwatch example to help teach GUI design and implementation, and more generally event driven programming design to our students. It illustrates many of the standard features and issues involved with MVC, including state based control and approaches to solving the coherence problem. Refactoring state diagrams into model state and view state is an important idea that is frequently overlooked.

Complete code for all the examples presented in this paper is available at `http://www.cs.uwp.edu/staff/hansen`.

5. REFERENCES

[1] Bruce, K.B., Danyluk, A.P. and Murtagh, T.P., *A Library to Support a Graphics-Based Object-First Approach to CS1*, SIGCSE Bulletin, 33(1), 2001, 6-10.

[2] Bruce, K.B., Danyluk, A.P. and Murtagh, T.P., *Event-driven Programming is Simple Enough for CS1*, Proceedings of the 6th Annual Conference on Innovation and Technology in Computer Science Education, Canterbury, England, 2001, 1-4.

[3] Burbeck, S., *Applications Programming in Smalltalk-80: How to use Model-View-Controller(MVC)*, University of Illinois in Urbana-Champaign (UIUC) Smalltalk Archive. Available at http://st-www.cs.uiuc.edu/users/smarch/st-docs/mvc.html, 1992.

[4] Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P. and Stal, M., *Pattern-Oriented Software Architecture: A System of Patterns*, John Wiley and Sons, New York, NY, 1996.

[5] Culwin, F., A *Java GUI Programmer's Guide*, Prentice-Hall, Upper Saddle River, NJ, 1998.

[6] Deitel, H.M., Deitel, P.J., Listfield, J., Nieto, T.R., Yaeger, C., Zlatkina, M., *C#: How to Program*, Prentice-Hall, Upper Saddle River, NJ, 2002.

[7] Deitel, H. M. and Deitel, P.J., *Java: How to Program, 5th ed.*, Prentice-Hall, Upper Saddle River, NJ, 2003.

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

[9] Rasala, R., Raab, J. and Proulx, V., *Java Power Tools: Model Software for Teaching Object-Oriented Design*, SIGCSE Bulletin, 33(1), 2001, 297-301.

[10] Wick, M., *Kaleidoscope: Using Design Patterns in CS1*, SIGCSE Bulletin, 33(1), 2001, 258-262.