Observer-Conditioned-Observable Design Pattern

By **Douglas A. Lyon** and **Carl Weiman**

ABSTRACT

The Observer-Conditioned-Observable (OCO) combines digitizing and transcoding of numeric change events. During the processing of numeric events, the transcoder converts the number from one range into another while preserving the mathematical integrity of the value. Numeric controllers that generate such events can be based in floating point numbers or integer numbers. For example, the *JSpinners* of Swing are based in floating point numbers, but *JSliders* are based in integers. Thus, the dynamic range of the *JSliders* is typically many orders of magnitude away from the dynamic range of the spinners. Even worse, changes in a numeric model can propagate to an integer-based spinner. This leads to quantization error and back propagation of the re-quantized value.

Interactive programs use multiple viewers and controllers to alter an underlying numeric model. These can update each other in an observer-observable loop that can propagate unintended and unmanaged digitization errors. The OCO design pattern breaks the loop and maintains control of the numerical values. The interception of changes is done via a modification of the *equals* method. If two numbers are *equal* (to within a user defined tolerance) propagation is suppressed. Thus, the digitizer re-samples and re-quantizes the numeric event.

Another potential problem arises when multiple viewer-controllers for the same number model employ differing scales. For example, sliders can range between two integer numbers (e.g., 0...100); whereas the floating-point number that we would like to model ranges from 0.0 and 1.0. We demonstrate a procedure for translating one value into the range of another, without error or feedback. OCO also samples events in time. This band limits the events to a level deemed reasonable for the application.

The veto design pattern is not new, nor, for that matter, is a numeric veto design pattern. However, a numeric veto design pattern that is sensitive to the magnitude and rate of the numeric change is new. Also new is the conversion of one numeric range into another during the model view controller construction. Thus, we have arrived at a new name for our design pattern, called the *Observer-Conditioned-Observable* design pattern (OCO).

 We apply the OCO design to the input of numbers that not only are floating point, but that have a dynamic range. The goal of our system is to allow the user to sweep large spans of dynamic range and "zoom in" to very high precision variation in areas of interest, for example bifurcation points in a dynamical system model. Our particular application involves 16 degrees of freedom for controlling the compound iteration of several hundred Java 3D linear transformations, in real-time. Other applications include scientific/mathematical simulation, Monte Carlo methods and numeric solution of systems of equations. These applications are critically dependent on initial conditions.
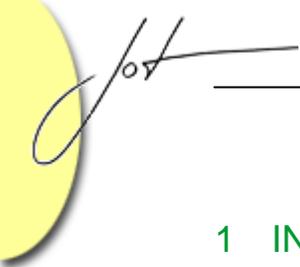
1   INTRODUCTION

The OCO design pattern ensures that consistency is maintained between normally incompatible numeric models, views and controllers. The numeric models intercept and convert numeric change events from integer-based controllers and from models that have different underlying numeric data types (i.e., conversion from floating point models to integer models, and back). The design is meant to provide a consistent method for allowing complex inter-numeric associations to communicate with integrity and without the danger of an endless observer-observable feedback loop.

In comparison, the intent of the traditional veto design pattern is to ensure that an instance of a numeric model does not exceed its range. It has no means of translating numeric values, nor does it allow for dynamic updates of dynamic range. The veto design pattern further lacks the feature of referring the communication between different numeric data types (like floating and fixed-point data types).

In the example that follows, we will show that the numeric resource provides an object-oriented means of translating ranges and enable the user to interactively set the increment used to change a variable. We will also show that a large variety of controllers can be used for both control and view. Finally, we present a example of a command design pattern to communicate the change events using values that are meaningful to the application programmers, hiding the internally held numeric models used to update views and controller settings. Further, we demonstrate that the OCO design pattern implementation is responsible for keeping track of translations, updates and conversions, freeing the programmer to attend to more substantive matters.

2   MOTIVATION

Visually interactive programs for mathematical models require GUI elements comprising the view and control of the model. The Model-View-Control design pattern incorporates these elements. In complex models, multiple GUI elements may refer to a single numeric quantity: a slider, textfield, gauge and spinner may all represent, for example, a single input voltage in a modeled circuit. GUI elements such as the Java *JSlider* class and the *SpinnerNumberModel* represent both view and controller; they are manipulated to change voltage, and they view changes broadcast to them from other programmed sources, typically via the Observer design pattern. Regardless, the underlying numerical model must be unique and all views should display the same quantity.

A problem arises when GUI elements with differing underlying number types interact. For example, Slider values are inherently fixed-point and *SpinnerNumberModels* can be floating point. If variable a updates variable b and variable b updates variable a, we have an update loop. The update loop will consume all the CPU resources of the given thread and the program will grind to a halt. Normally, we break the observer-observable update
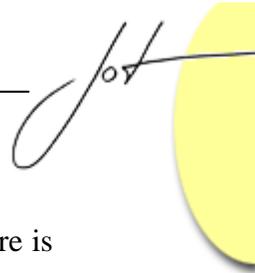
loop by determining if the event creating the update represents a change of state. If there is no change, we do not propagate the update.

This works fine when the numerical types of variables *a* and *b* are commensurate; they both settle to the same state and no change event is emitted. But if a floating point variable triggers a fixed point observer, the latter will introduce a digitization error and return a new (state changed) value to the former, which in turn, returns a new value to the latter, and so on.

A better solution is to make a design pattern that is responsible for keeping track of the instances created from numeric models. The pattern manages publish-subscribe relationships and provides a consistent means of controlling updates (with parametric control over precision). The new design pattern is called the OCO design pattern and it provides a way to link multiple controllers and viewers to a numeric quantity.

3 APPLICABILITY

Use the OCO design pattern when:

1. Fixed-point controllers must control floating-point variables.
2. Variables with different ranges must track each other.
3. Floating point variables must control fixed-point variables.
4. Floating point precision must be considered during the update process.

4 STRUCTURE

Observer-observable control system
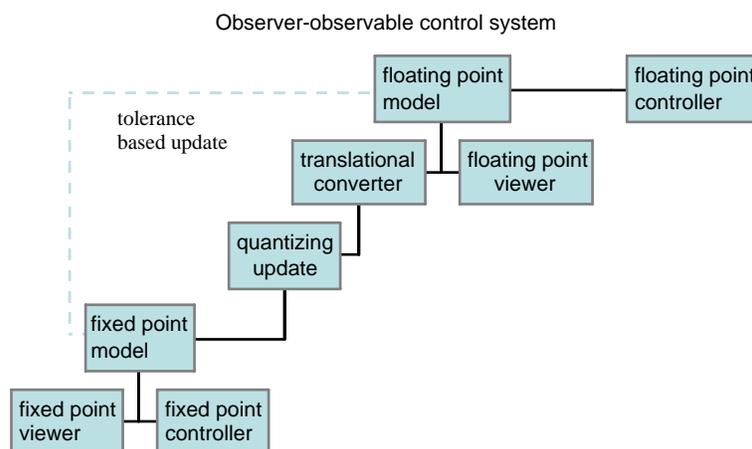
Figure 4.1 A diagram for the OCO design pattern.

Figure 4.1 shows a floating-point model with a standard MVC design [Cooper]. A translational converter takes the model, alters its range, increment and type, creating a quantized version of the model, suitable for the fixed-point representation. The fixed-point
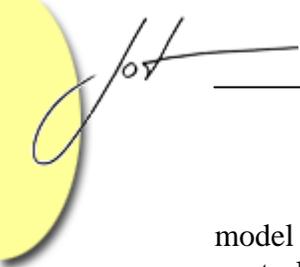
model is subjected to a similar MVC design, using a fixed-point view and fixed-point controller. However, to keep the systems from back propagating to the floating-point model, a tolerance is introduced. Thus, updates from the fixed-point model to the floating-point model are filtered so that they exceed a given tolerance. This prevents observer-observable feedback leading to "dither" around the integral set point.

We use switching control in place of dither to improve the quality of our state variable tracking. By placing bounds on the tracking error, we create a boundary layer so that we control high frequency chatter in our MVC designs.

The tolerance that we have selected is set to a constant that may be changed by the user. However, this presents itself as a possibly slowly varying or uncertain parameter that gives rise to adaptive control. Additionally, we have not band limited the update rate and thus numeric tolerance is independent of time, making our present implementation autonomous.

5 PARTICIPANTS

The participants are the OCO design pattern clients that need consistency to be maintained between different numeric data types.

The OCO design pattern updates an instance when a change occurs, if, and only if, the change is of an order of magnitude larger than some given amount. The pattern propagates the change after translation into the target variable's range. The pattern is responsible for keeping track of the observers and their unique properties (range, fixed vs. floating point, increment, tolerance for change, etc.).

6 COLLABORATIONS

Clients obtain a reference to a numeric model and add themselves as observers of this model. If the client has its own numeric model, then the mediator links the two models using a publish-subscribe relationship. The rules of the update between the two models are made explicit at mediator link time. If a numeric instance is left in an improper state (e.g., the numeric state variables are out of sync with one another) it is NOT the role of the mediator to propagate the updates, but rather the role of the numeric models to update one another, according to the update rules.

Further, it is not the role of the OCO Pattern to manage the creation of numeric model resources. That is, multiple numeric resource creations can be had as a side effect of the creation of new controllers and views. These can be linked, via the mediator or be delegated to some other part of the system of the system.

7 CONSEQUENCES

The OCO Design Pattern has several benefits:

1. The OCO Design Pattern controls updates between numeric instances. The pattern uses translates numeric values into the correct range before propagating the value.
2. Centralized and hidden complexity. The pattern avoids distributing numeric business logic throughout the application, centralizing the translation of valued between variables as well as the policy for updating them.
3. Ensure consistent variables. The pattern maps valued from one variable into another uses the publish-subscribe relationship.
4. **Variable precision setting.** The pattern allows for variable precision on the input spinner, to enable the user more control over variable increments

8 IMPLEMENTATION

Here are implementation issues to consider when using the OCO design pattern:

1. Invertability of the mapping of values. The OCO design pattern requires that there be a means to map a variable's value into a new parameter space and that the value be able to be mapped back (to within an given error).
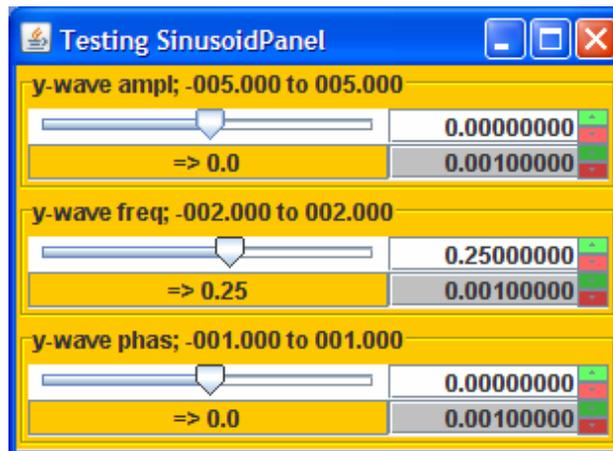

Figure 8.1-1. OCO GUI for control/view of sinusoid parameters.

Figure 8.1-1 illustrates the broad range and high precision of the OCO GUI. Each panel shows the GUI of one of the three parameters representing a sinusoid. The sliders span the entire range of the parameters. The lower spinner (gray background text field) defines the size of the increment of the upper spinner. The upper spinner refers to the parameter itself.

The button under the slider resets the value of the parameter to its initial value, a valuable tool when exploration of parameter space leads to dead ends.
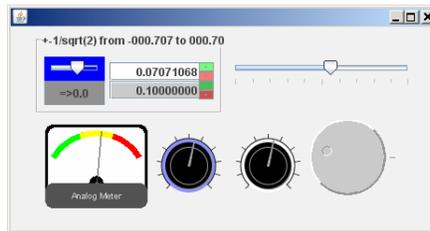

Figure 8.1-2. Multiple Views and Controllers

Figure 8.1-2 shows multiple viewers and controllers for a single number model. The increment on the number is controlled using the lower spinner and moves in powers of 10. A reset button appears below one slide, and this resets the default value to 0. The spinners respond to up and down keyboard or mouse arrow input by changing the number model in a manner consistent with the increment. Page-up and page-down keys are used to change the number model faster. Floating-point to fixed-point conversions are handled internally by those views and controllers that require it. A new number model, that contains semantic information about the number (range and increment) is used to keep the viewer and controllers properly bound:

```
public static void main(String[] args) {
        RunNormalizedSpinnerNumberModel snm =
                new RunNormalizedSpinnerNumberModel(0,
                        -1 / Math.sqrt(2), 1 / Math.sqrt(2),
                        0.001) {
                    public void run() {
                    }
                };
        ClosableJFrame cf = new ClosableJFrame();
        cf.setContainerLayout(new FlowLayout());
        Container c = cf.getContentPane();
        c.add(RunSliderDouble.getRunSliderDouble(snm, "+-1/sqrt(2)"));
        c.add(RunPercentageSlider.getRunPercentageSlider(snm));
        c.add(AnalogMeter.getAnalogMeter(snm));
        c.add(RunKnob.getRunKnob(snm));
        c.add(RunKnob.getRunKnob(snm));
        c.add(RunShuttle.getRunShuttle(snm));
        cf.pack();
        cf.setVisible(true);
    }
```

The getter factory methods have use the OCO design pattern to link the external number model with the internal number model. For example, the *RunShuttle* was design to work with numbers that range from zero to one. In order to provide a proper scale, the factory method uses the OCO design pattern for scaling and limiting propagation of feedback:

```
public static RunShuttle getRunShuttle(
      RunNormalizedSpinnerNumberModel rnsnm) {
         final RunNormalizedSpinnerNumberModel
      zeroToOne = new RunNormalizedSpinnerNumberModel(0, 0, 1, 0.1) {
            public void run() {
            }
         };
         // OCO DP
         rnsnm.publishToSpinnerNumberModel(zeroToOne);
         zeroToOne.publishToSpinnerNumberModel(rnsnm);
         return new RunShuttle(zeroToOne) {
            public void run() {
            }
         };
      }
```

Run methods are used for all components, in a manner consistent with the *Imperion* project [Lyon].

9   SAMPLE CODE - THE VERNIER

We are presented with two interfaces, a slider and a spinner. The slider allows easy access to a large range of numbers, but with low precision. The spinner allows slow access to a small range of numbers, but with high precision. Thus, we observe that precision is inversely related to speed and range of access, in these components. The sliders access of numbers lacks precision because it uses a fixed-point increment (limiting its dynamic range). This is due, in part, to the problem of the resolution of the mouse and its so-called mouse coordinate system. The spinner, on the other hand, has no such limit, as a number with 64 bit precision may be typed in to its text field. However, the up and down arrows limit the precision with which the spinner may sweep through a range of number. To alter the step-size used during the increment and decrement actions, we add another spinner, which we call the vernier. The following code uses the OCO design pattern to create several controllers and viewers for a 16 dimensional 3D compound iterated system.
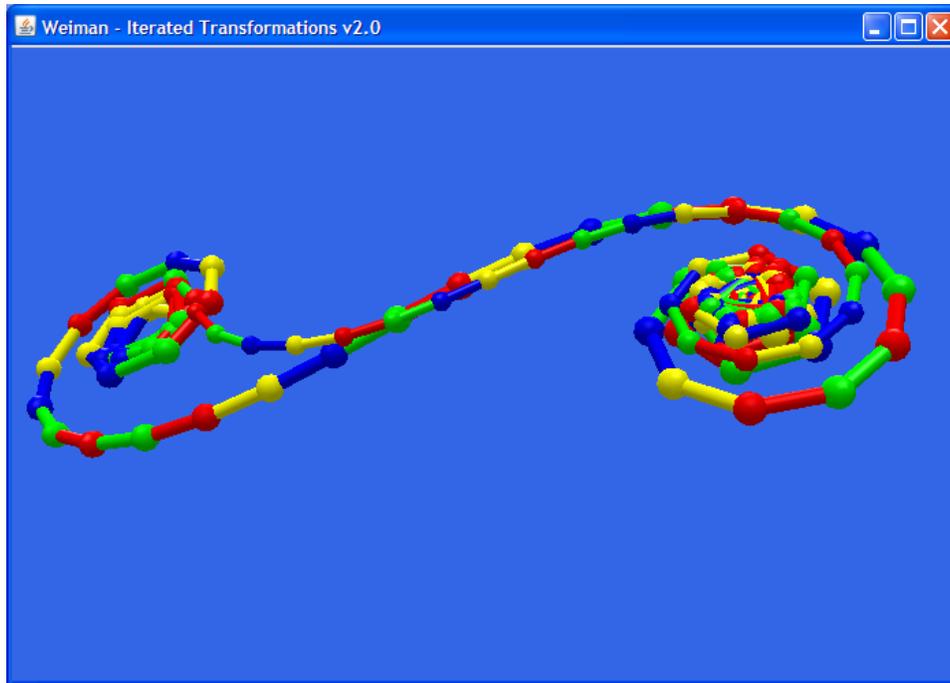
Figure 9.1-1. Trajectory of non-linear system in state space.

Figure 9.1-1 shows the behavior of an iterated system of transformations (rendered with Java3D) whose 15 parameters are interactively controlled by the user via OCO control. The behavior is critically dependent on initial conditions, necessitating both the wide range and vernier control provided by the OCO GUI.

10 RELATED WORK

The veto, observable and mediator design patterns are not new [Gamma et Al.]. However, the OCO design pattern is new, as far as we know. While both the mediator and observer-observable design patterns are association patterns, the observer-observable ensures that the update instance is always propagated [Goldfedder]. In comparison, the OCO ensures that the update is only sent if the change is greater than a given magnitude.

There are other extensions to the observer-observable design pattern, such as vetoable change listener (a listener that can overrule property change events). *However*, the vetoable change listener is not generally applied to numeric events (particularly with a precision) [Allen] [Grand].
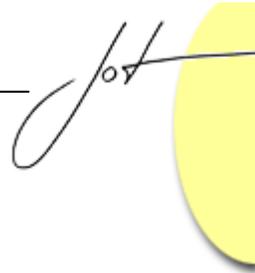
11 SUMMARY

This paper presents a solution to multiple coupled number models of different type and different range. Digitization and observer design pattern are coupled to provide a design that controls feedback rates and re-quantized signals. The OCO's role in signal processing is to transcode, quantized and re-sample (i.e. digitize and condition) the signal.

We are concerned that the OCO design pattern may create zombie objects (i.e., objects that are never reclaimed by the garbage collector). This can lead to memory leaks [Javatip 79]. We have yet to demonstrate this problem in our implementations, and, should it arise, the question of how to address it remains open.

REFERENCES

[Allen] Mitch Allen and John B. Harvie. *Hands on Java Beans*. Prima Publishing, 1997.

[Beck97] Kent Beck. *Smalltalk Best Practice Patterns*. Prentice Hall, 1998.

[Cooper] James W. Cooper. *Java Design Patterns*. Addison-Wesley, 2000.

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

[Goldfedder] Brandon Goldfedder. *The Joy of Patterns*. Addison-Wesley, 2002.

[Grand] Mark Grand. *Patterns in Java, Volume 1*. John Wiley & Sons, Inc. 1998.

[Java Tip 79] Raimond Reichert. "Java Tip 79: Interact with garbage collector to avoid memory leaks. Use reference objects to prevent memory leaks in applications built on the MVC pattern" JavaWorld.com, 10/20/99. http://www.javaworld.com/javaworld/javatips/jw-javatip79.html Last accessed March 26, 2007.

[Lyon] Douglas Lyon. Project Imperion: New Semantics, Facade and Command Design Patterns for Swing, *Journal of Object Technology*, vol. 3, no. 5, May-June 2004, pp. 51-64. http://www.jot.fm//issues/issue_2004_05/column6
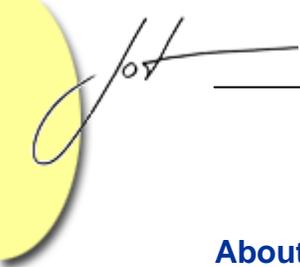
About the authors

Douglas A. Lyon (M'89-SM'00) received the Ph.D., M.S. and B.S. degrees in computer and systems engineering from Rensselaer Polytechnic Institute (1991, 1985 and 1983). Dr. Lyon has worked at AT&T Bell Laboratories at Murray Hill, NJ and the Jet Propulsion Laboratory at the California Institute of Technology, Pasadena, CA. He is currently the Chairman of the Computer Engineering Department at Fairfield University, in Fairfield CT, a senior member of the IEEE and President of DocJava, Inc., a consulting firm in Connecticut. Dr. Lyon has authored or co-authored three books (Java, Digital Signal Processing, Image Processing in Java and Java for Programmers). He has authored over 30 journal publications. Email: lyon@docjava.com. Web: http://www.DocJava.com.

Carl Weiman is adjunct Associate Professor at Fairfield University where he teaches courses in Java including Voice and Signal Processing and 3D Computer Graphics in the Computer Engineering Department. He has also been visiting professor at the Cooper Union for the Advancement of Science in Art, team-teaching an interdisciplinary course on Robotics and Theater, sponsored by an NSF grant of which he was PI.

In his previous career as Director of Research at HelpMate Robotics, he won $2M in grants from NASA, NSF and DoD for robot vision and navigation. This work led to 9 patents and numerous publications. Prior to that he was Systems Engineer in GE flight simulation systems, designing graphics algorithms. His Ph. D. thesis from Ohio State University was titled "Pattern Recognition by Retina-Like Devices". A driving force in all his work is visualization and algorithms for modeling neurophysiological mechanisms of biological vision.