Case Study

Exposing internal details to the UI

(Indicating to the user when a signal level threshold has been exceeded)

Steve Dwyer

Since completing my Certificate of Higher Education in the Mathematical Sciences, I've been working through Will Pirkle's book *Designing Audio Effect Plugins in C++*. I've since completed a number of effects pedals.

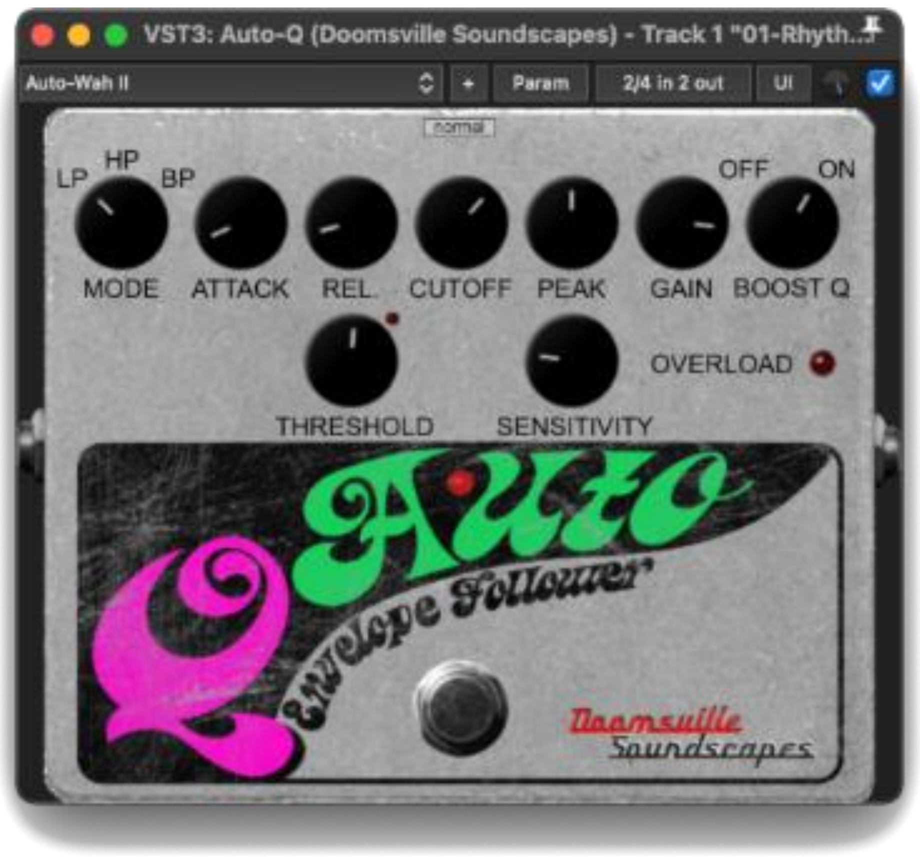
I would like to invite you to visit my GitHub repository https://github.com/DoomyDwyer/ASPiKProjects where my work is available for review. Each project uses classes from files maintained in a second repository, which contains common classes used across all of the effects pedals, and is available here: https://github.com/DoomyDwyer/ASPiKCommon. This is where I further worked on the exercises from the book, to give the effects my own flavour, and change the code to suit my needs and tastes.

The plugins, available as VST3 and Audio Units, as well as presets and user guides, can be downloaded here:

https://github.com/DoomyDwyer/ASPiKProjects/releases

A number of these effects, such as the *Auto-Q* Envelope Follower and the *Memento* "Ducking" Delay, use an *Envelope Detector* to measure the amplitude of a signal. The value returned by the Envelope Detector can then be used to drive further logic in the Digital Signal Processing algorithm of the effect. One common control, represented as a knob on the effects GUI, which might be used in an algorithm using an Envelope Detector, is a *Threshold* value. The DSP algorithm may alter its processing, depending on whether the threshold has been exceeded or not. Depending on the values of other knobs' settings on the GUI, the sound produced by the effects could be altered significantly or only very subtly, depending on whether the threshold has been exceeded or not.

I decided that, to help the user dial in a suitable value for the threshold when processing any given signal, a visual cue would be very useful, which indicates whether or not the threshold has been exceeded at any given point in time. A LED which lights up when the threshold has been exceeded would be a simple visual cue, which also aligns with my principle that effects plugins which have a GUI resembling an analog effects pedal should consistently resemble visually and behave similarly to a traditional analog pedal. In the below screenshot of the *Auto-Q* Envelope Follower effect, at the top right of the THRESHOLD knob, there is a small LED, which should then light up whenever the detected amplitude envelope level exceeds the threshold value:



This is all fairly straightforward. The problem then arises, how do we communicate this change of state (threshold exceeded or not exceeded) to the UI of the effect, when we're constrained by the following:

- the DSP algorithms are in theory components which could be reused in more than one effects plugin, so can't be tied to any control on the UI. Moreover,
- these DSP components don't know anything about the plugin in which they're being used.

The DSP components are contained as dependencies within the plugin, but they have no access to their dependent, the plugin itself.

So how do we notify the plugin of state changes within the Envelope Detector, while maintaining the loose coupling described above?

I opted to use the GoF (Gang of Four) Design Pattern *Observer*[1]

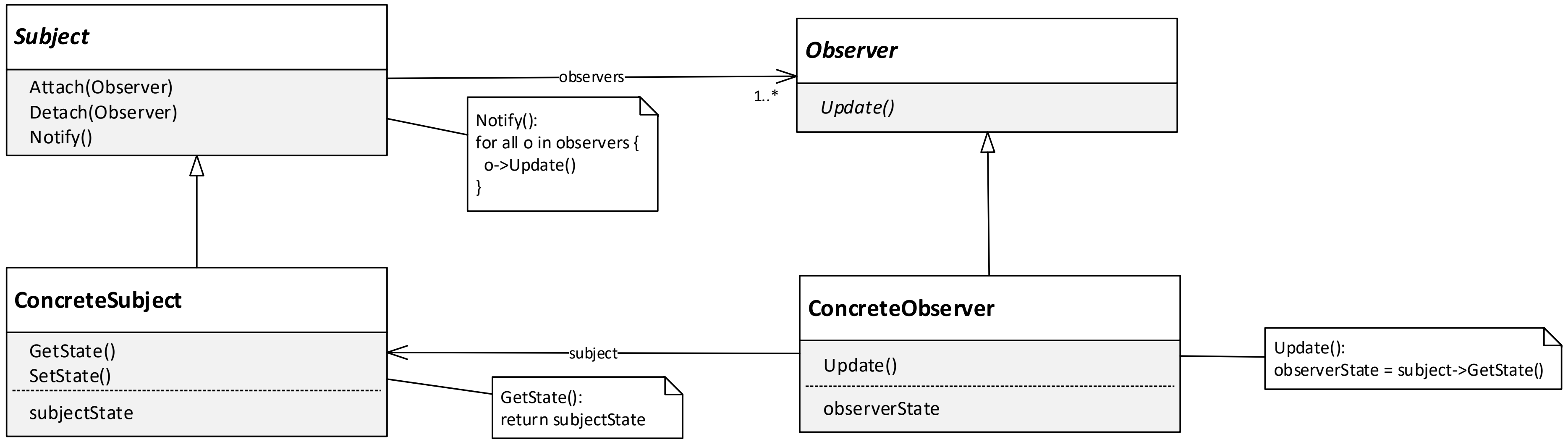
The GoF Design Pattern *Observer* has as its *Intent*:

- *Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.*

One core design principle of this pattern, is that the loose coupling between these objects is maintained.

[1] *Design Patterns: Elements of Reusable Object-Oriented Software*, pages 293-303. Gamma, Helm, Johnson, Vlissides. Addison-Wesley, 1995. ISBN 0-201-63361-2

The class diagram of the generic Observer structure, as defined by the Gang of Four, is as below:



The `Attach()` and `Detach()` methods are used to *subscribe* or *unsubscribe* an observer with its Subject. The `Notify()` method *publishes* notifications to its observers, without needing to know anything about its observers, other than that they support the *Observer* interface. Furthermore, Observers can subscribe or unsubscribe to a Subject's update notifications at runtime, thereby maintaining the loose coupling of the various components within the plugin.

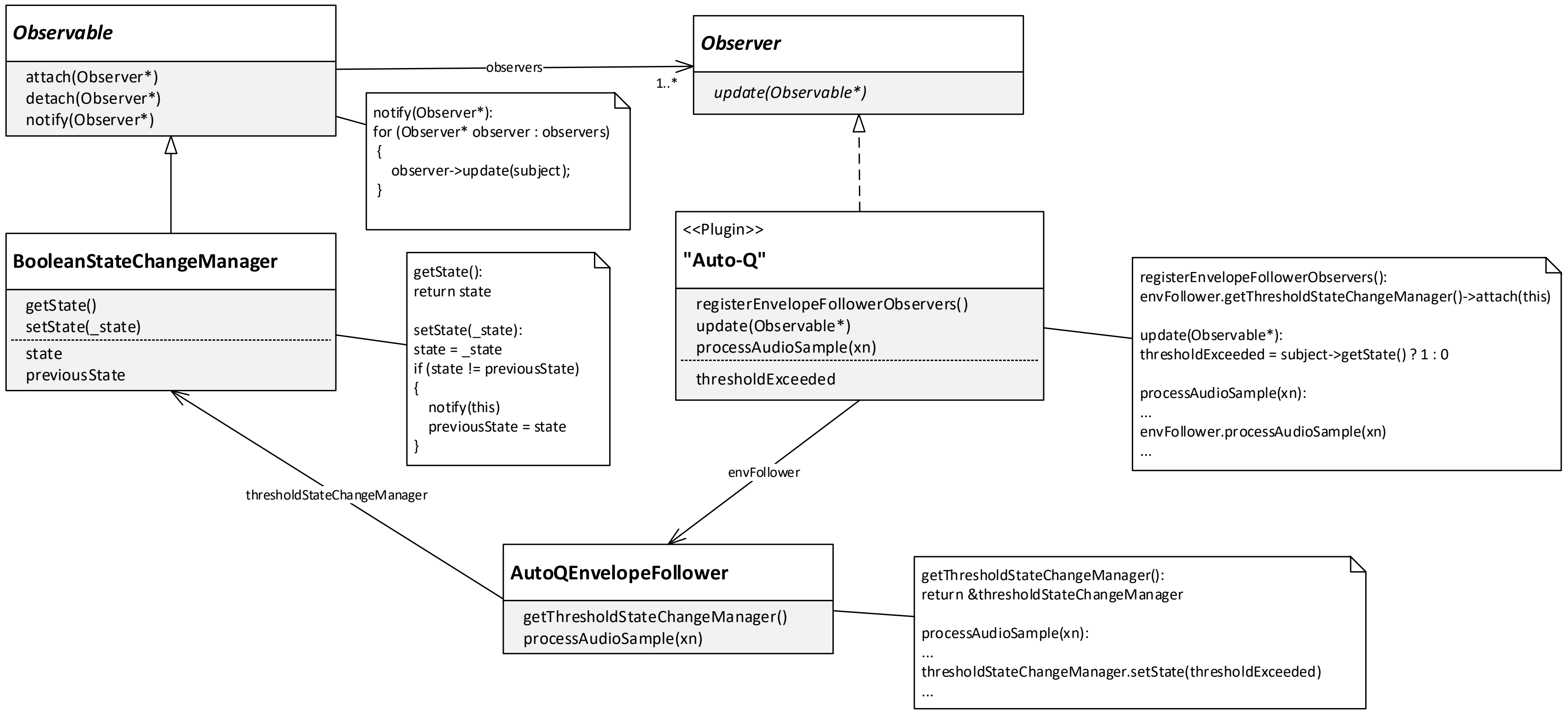
Moving on from the generic pattern to the implementation used within the plugins; within the plugin itself, the work required to add a new LED is rather trivial; a new variable is added to the plugin, *`thresholdExceeded`*, which essentially acts as a VU meter, which can take one of two values; 0 or 1. This variable is bound to the LED on the GUI, which will be updated in a thread-safe manner on the GUI Thread.

An `Observable` class (the "Subject") and an `Observer` interface are both written; The plugin itself implements the `Observer` interface and therefore must implement the `update()` method (within the plugins we've adopted camelCase, which diverges from the style used in the Gang of Four book). Since we're dealing with Boolean state changes within this Use Case (the threshold has either been exceeded, or not), a further specialisation of the `Observable` class is also created: the `BooleanStateChangeManager` class. This implementation stores its `state` (the "subjectState") as a bool. It also caches its previous state, so that the `notify()` method is only executed when the current state differs from the state recorded the last time `notify()` was invoked. This prevents a flurry of notifications, which would otherwise be generated for each sample processed.

A further enhancement has been added above & beyond what is described in the generic Observer Design Pattern, an idea borrowed from Java's Observer/Observable class design; the "Subject" (in this case the `BooleanStateChangeManager`) passes itself as a parameter when calling each subscribed observer's `update()` method. This allows the Observer to not necessarily be tied to just one Observable. Although in these plugins there is currently only one Observable in use, I consider this a better design. The issue of the generic Design Pattern only supporting one "Subject" is also discussed in the GoF Book, so this has clearly been recognised as a flaw in the original design. When the `update()` method on the Observer is invoked, since the "Subject" has been passed as a parameter, the Observer has access to all the public members of that object, `getState()` being the one of interest to the Observer.

During plugin initialisation, the plugin creates a member object containing the Envelope Detector (the *Auto-Q* Envelope Follower effect has an individual Envelope Detector for each audio input channel, but we can ignore that detail when only concerned with the aspect of the design currently under discussion here) and then registers itself as an observer to this object containing the Envelope Detector. The member object containing the Envelope Detector also contains an instance of `BooleanStateChangeManager` (the "Subject", a subclass of Observable) and the registration basically invokes this object's `attach()` method, passing in a reference to the plugin (an implementation of `Observer`), thereby subscribing the plugin to notifications of the "Subject's" change of state.

The class design of this implementation of the Observer Design Pattern is as below (the design has been simplified by only showing one Envelope Follower, rather than having one per audio input channel):



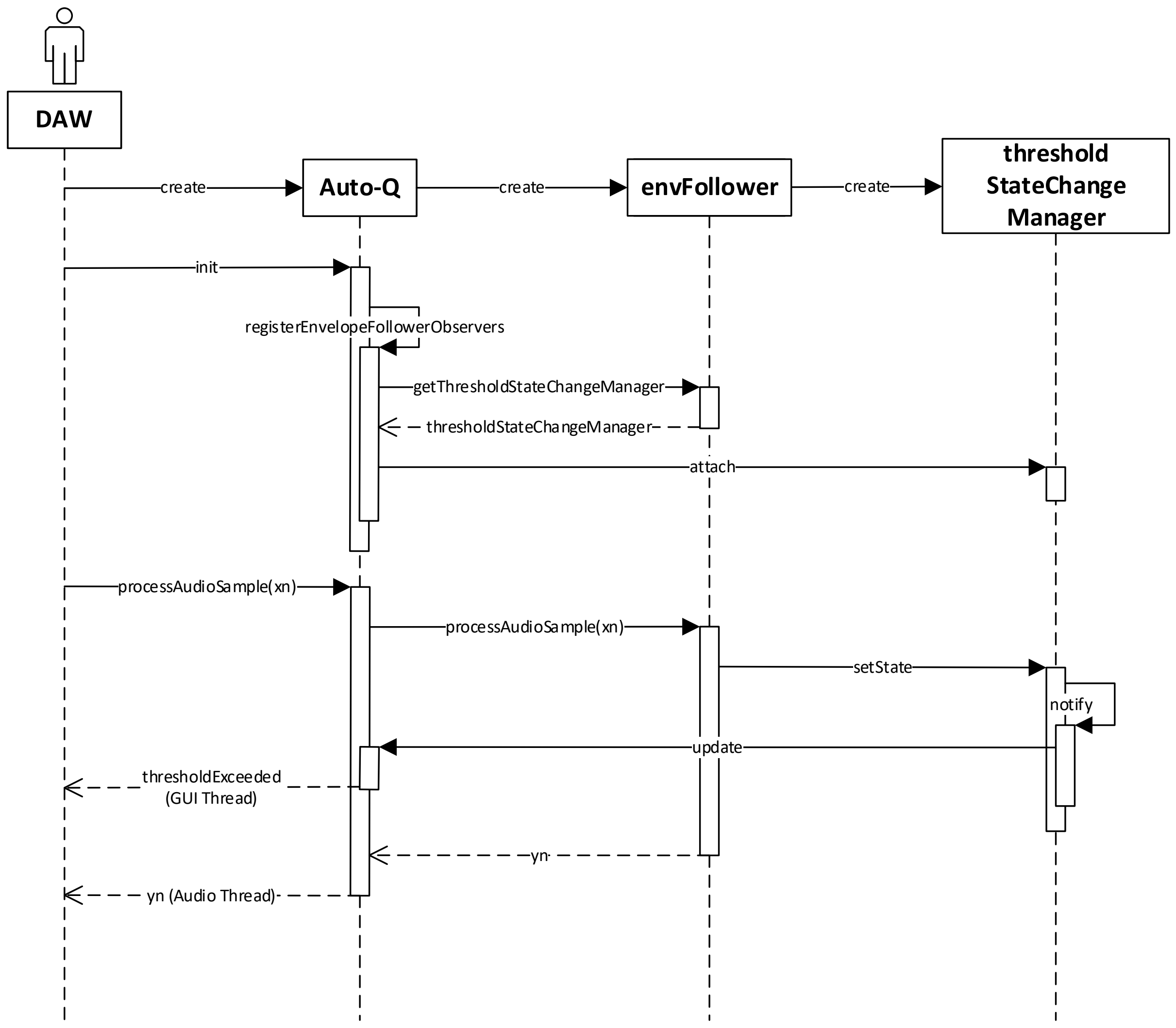
```
notify(Observer*):
for (Observer* observer : observers)
 {
     observer->update(subject);
 }
```

```
getState():
return state

setState(_state):
state = _state
if (state != previousState)
{
   notify(this)
   previousState = state
}
```

```
registerEnvelopeFollowerObservers():
envFollower.getThresholdStateChangeManager()->attach(this)

update(Observable*):
thresholdExceeded = subject->getState() ? 1 : 0

processAudioSample(xn):
...
envFollower.processAudioSample(xn)
...
```

```
getThresholdStateChangeManager():
return &thresholdStateChangeManager

processAudioSample(xn):
...
thresholdStateChangeManager.setState(thresholdExceeded)
...
```

The interaction between the components is as follows:


DAW
Auto-Q
envFollower
threshold StateChange Manager
create
create
create
init
registerEnvelopeFollowerObservers
getThresholdStateChangeManager
thresholdStateChangeManager
attach
processAudioSample(xn)
processAudioSample(xn)
setState
notify
update
thresholdExceeded (GUI Thread)
yn
yn (Audio Thread)