The Observer Pattern

Relevant Links

- OODesign[1]
- GoF Book[2], page 293
- Backbone's Events Implementation[3]

Notes

We start our survey of Patterns with the Observer Pattern.

Classification

The Observer Pattern is a Behavioral Pattern.

It is also known as Publish/Subscribe when used on a "global" Event object.

Motivation

We often have situations when a varied number of objects need to become aware a change in an object's state, without enforcing a very strong linkage between the two.

For instance in our TODOApp there is a list that maintains our task items. When that list changes, or when items in that list change in some way, a number of UI elements might need to change in response to that, or some database backend may need to be updated and so on. On the other hand, the task list should not know too much about all those different parts of the program that may need to know about its changes. It should however offer them a way to keep themselves apprised of any changes.

Intent

More abstractly, the intent of the Observer Pattern is to define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified automatically.

[1] http://www.oodesign.com/observer-pattern.html

[2] http://www.amazon.com/Design-Patterns-Elements-Reusable-Object-Oriented/dp/0201633612/

[3] http://backbonejs.org/docs/backbone.html#section-16

Participants / Implementation

The Observer Pattern consists of the following, with slight variations depending on the target language. We first describe it in the more abstract form and with a statically typed language in mind, then we discuss its implementation in Javascript.

Subject or **Observable**, is an abstract interface/class that provides for the following methods: - registerObserver to add a new observer. - unregisterObserver to remove a given observer. - notifyObservers to inform the observers of the change. - As a class it would further contain an observerCollection.

ConcreteObservable is a concrete implementation/subclass of Observable, that maintains some state variable, and in its setState method calls notifyObservers.

Observer is an abstract interface/class that provides for a single notify method. When notifyObservers is called, it goes through each observer on its list and calls the observer's notify method.

ConcreteObserver classes that implement/subclass Observer must implement their own notify method that handles their needs.

Javascript implementation In Javascript implementations, usually done via an "Events" class/mixin, there are two key differences:

- There is no explicit observer interface to adhere to. Instead, observers pass the function they want to have called into the registerObserver method.
- "Events" usually have a "topic" associated with them. So observers can register to listen to only "mouseup" events for example, rather than "all events". In other words, an Observable could accept observers for different "topics".

The interface provided by events typically looks like this:

on obj.on(topic, handler, [context]) registers the function handler to respond to events associated to the string topic. If context is set, it is used as the this when the handler is called.

off obj.off(topic, handler, [context]) deregisters the function handler from the topic, and only if it's attached to the specific context (or null).

trigger obj.trigger(topic, arguments...) calls the handlers associated with the topic, and passes all remaining arguments to them.

once A convenience method that would attach a handler that deattaches itself after its first invocation. We will not implement this, but some implementations have it there, along with some other convenience methods.

We will see later details on how to implement this. Internally, we will mix in the "Events" class to any class that we want to make Observable. A hidden variable named _events will hold the information about all the various handlers that the object needs to keep around.

An implementation can be found here[4].

One important consideration is regarding when the handlers should be fired. You have to choose in your implementation whether they should fire right away, and only then return control to the function that triggered the event, or whether they should set a timer for them to fire at the first available moment after the current function has finished running.

Example Here's a simple example of a timer that sends a notification out every minute to whoever's listening on the global Event channel.

```
(function(){
    var i = 0;
    function callem() { i+= 1; Event.trigger('tick', i); }
    setInterval(callem, 60000);
}());
```

The most standard application of the Observer model is in the Model-View-Controller pattern we will discuss later. Models are responsible for application state, and they send out notifications whenever their state changes.

Similarly, and related, a Collection could send notifications out when a new item is added, and it can also listen to changes on its items and forward them to its observers, so that they don't have to observe the items themselves.

[4] ../testPages/events.js