Custom Events

Relevant Links

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

Notes

We will discuss here how to create custom events, in the context of discussing an important *design pattern*, the **observer pattern**, along with the closely related **publish/subscribe** pattern.

Problem description

Design patterns in general aim to offer a standard, tried-and-true, approach to solving a design problem. They also become a standard way for programmers to communicate intent.

In this case the problem is that of enabling modules to communicate with each other without a very direct link to each other (so keeping the modules loosely coupled). Both systems effectively allow one module to be "notified" when something happens to another module.]

So we can have an "observer" that is notified when something happens to another "subject". The key idea is that the subject does not need to know exactly who is observing it. We can write the code for the subject without caring who, if anyone, is observing.

Example

As an example, imagine the task list we are working on. When a task's title changes, the interface must be updated. How will the interface know that it must be updated?

That depends on what caused the update. The typical case would be that the user had some option on the interface to change the title. Then when that is done the interface knows that the title will change, so it knows that it must update itself.

But what if it is possible that the tasks change remotely, from say a collaborator who is allowed to also edit the same tasks from his computer. How will our interface then know that it must be updated?

[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

One approach is for the model to directly know all the other parts that need to change when it changes. For instance in the code that updates the model's title, we might have something like:

```
function changeTitle(newTitle) {
  // Change the title
  this.title = newTitle;
  // Notify all "users"
  interfaceController.updateTitle(this);
  database.updateTitle(this);
  ... more calls to other parts that need to know?
}
```

This is hard to maintain, as it forces each object to be *aware* of all these other components. And if a new component is added in the future, we now need to search through the code and find all the parts that need to be altered.

And what if I want to test the Task class and this changeTitle method? While testing, maybe I don't care about the interface and the database, I just want to test the objects themselves. The way the code above is written I cannot do that: Any call to changeTitle will also call an interfaceController method, a database method, and who knows what else.

This is the problem that the observer pattern aims to avoid.

The pattern

The general approach is the following:

- Objects *trigger* an event when something noteworthy happens.
- Other objects can *register* to be called when an event is triggered.
- When the event is triggered, all objects that register for it get notified via a function they provided.
- If no objects registered, nothing happens.

There are two versions of this technique:

- The **observer pattern**, where objects can register to trigger a function when some "event" happens to another object. As a real-life example, we advisors have effectively registered with the registrar's office to receive an email if a student changes advisors. This is not something that the student needs to act on in any particular way. It just happens.
- The **publish/subscribe pattern**, which effectively uses a single global communications system: You can *subscribe* to listen to an event with a given "name". Then you get notified when any other part of the application *publishes* an event with the given name. Imagine somehow being able to tell your email system to "send me a link whenever any news articles show up that mention Computer Science and my name".

More formally, we could say that *the Observer Pattern defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified automatically.*

Javascript implementation Javascript implementations of this pattern are usually done via an “Events” class/mixin. You typically would be given the following:

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. The square brackets here mean that the third argument is optional, and not that it should be an array.

topic here is basically just a string. An object could trigger many different events with different topics. And handlers must register for the one they care about.

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. It is really the latter that should happen.

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);
}());
```

[4] ../../testPages/events.js

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.