The Observer Pattern

Relevant Links

- OODesign[1]
- GoF Book[2], page 233

Notes

The Command Pattern is used to encapsulate requests.

Classification

The Command Pattern is a Behavioral Pattern.

Motivation

Some times we need to issue requests to objects without knowing anything about the requested operation or the request's receiver. For instance in a web application, clicking on a button or many item should trigger some operations in the background, like downloading a new resource or something of the kind. The specific code that is responding to the button click does not need to know exactly what that operation is, or what object it is a part of. It just needs to know that it is supposed to execute it.

Similarly, a "history" component in a text-editing application, that keeps track of changes so they can be reverted, does not need to know exactly what these changes are, it just needs to have a way to undo or redo them.

Intent

The Command Pattern encapsulates a request as an object, thereby allowing us to parametrize clients with different requests, log requests, or support undoable operations.

The Command Pattern decouples the object that invokes the operation from the one that knows how to perform it.

In functional / procedural languages, the command pattern is often implemented as simply a *callback* function.

The setTimeout and setInterval functionalities are good examples of the Command Pattern. They just expect a function to execute, and they do not need nor care about any particulars of the function.

[1] http://www.oodesign.com/command-pattern.html
[2] http://www.amazon.com/Design-Patterns-Elements-Reusable-Object-Oriented/dp/0201633612/

Participants / Implementation

In a statically typed language, the Command Pattern requires the following pieces:

Command an interface containing a single execute method, or optionally an unexecute method if we want to include undoable commands. We will extend it a bit to also include an isUndoable boolean.

ConcreteCommand a concrete class that implements the Command interface and has a specific functionality in mind (e.g. copying, pasting, opening a file etc).

ConcreteCommand defines a binding between a receiver object and an action to be performed, and implements execute by performing the necessary actions on the receiver.

Client (usually the application) creates the specific ConcreteCommand object and sets its receiver.

Invoker (e.g. the UI element) is the one asking the command to call execute. This could be a menu item or its handler, etc.

Receiver knows how to handle the operations included in the request.

We can also assemble commands into a composite command (strictly speaking an example of the Composite pattern).

Javascript implementation In Javascript this could be implemented as a simple function, that gets called as a means of "execute". If more functionality is needed, then a simple object with an execute method, and possibly an unexecute method, would do.

Example Here is a simple example that creates a menu list with actions bound to each element.

```
function makeList(items) {
    // Each item has "title" and "command".
    var ul = document.createElement("ul");
    items.forEach(function(item) {
        var li = document.createElement("li");
        li.innerHTML = item.title;
        li.addEventListener('click', items.command.execute);
        ul.appendChild(li);
    });
    return ul;
}
```

Here is how we could build a simple macro command that is meant to execute a series of commands:

```
function makeMacro(commands) {
    var o = {};
    o.commands = commands || [];
    o.addCommand = function(command) { commands.push(command); }
    o.execute = function() {
        commands.forEach(function(cmd) { cmd.execute(); });
    }
    return o;
}
```

We will do an example that combines observer and command to manage a list of items.

```
var List = newClass(function init() { this.values = []; });

Event.mixin(List.prototype);

List.prototype.add = function(v) {
    this.values.push(v);
    this.trigger('add', v);
    return this;
};

var lst = List.new();
var ul = document.createElement("ul");
var input = document.createElement("input");

lst.on('add', function(v) { console.log("Added: ", v); });

document.body.innerHTML = "";
document.body.appendChild(ul);
document.body.appendChild(input);

// Command that adds list element based on input contents
var cmd = {
    execute: function addElement() { lst.add(input.value); }
};

input.addEventListener('change', cmd.execute);
lst.on('add', function(v) {
    var li = document.createElement("li");
    li.innerHTML = v;
    ul.appendChild(li);
});
```