Chapter 79: Behavioral Design Patterns

Section 79.1: Observer pattern

The Observer pattern is used for event handling and delegation. A *subject* maintains a collection of *observers.* The subject then notifies these observers whenever an event occurs. If you've ever used `addEventListener` then you've utilized the Observer pattern.

```
function Subject() {
    this.observers = []; // Observers listening to the subject

    this.registerObserver = function(observer) {
        // Add an observer if it isn't already being tracked
        if (this.observers.indexOf(observer) === -1) {
            this.observers.push(observer);
        }
    };

    this.unregisterObserver = function(observer) {
        // Removes a previously registered observer
        var index = this.observers.indexOf(observer);
        if (index > -1) {
            this.observers.splice(index, 1);
        }
    };

    this.notifyObservers = function(message) {
        // Send a message to all observers
        this.observers.forEach(function(observer) {
            observer.notify(message);
        });
    };
}

function Observer() {
    this.notify = function(message) {
        // Every observer must implement this function
    };
}
```

Example usage:

```
function Employee(name) {
    this.name = name;

    // Implement `notify` so the subject can pass us messages
    this.notify = function(meetingTime) {
        console.log(this.name + ': There is a meeting at ' + meetingTime);
    };
}

var bob = new Employee('Bob');
var jane = new Employee('Jane');
var meetingAlerts = new Subject();
meetingAlerts.registerObserver(bob);
meetingAlerts.registerObserver(jane);
meetingAlerts.notifyObservers('4pm');

// Output:
```

```
// Bob: There is a meeting at 4pm
// Jane: There is a meeting at 4pm
```

Section 79.2: Mediator Pattern

Think of the mediator pattern as the flight control tower that controls planes in the air: it directs this plane to land now, the second to wait, and the third to take off, etc. However no plane is ever allowed to talk to its peers.

This is how mediator works, it works as a communication hub among different modules, this way you reduce module dependency on each other, increase loose coupling, and consequently portability.

This **Chatroom example** explains how mediator patterns works:

```
// each participant is just a module that wants to talk to other modules(other participants)
var Participant = function(name) {
    this.name = name;
    this.chatroom = null;
};
 // each participant has method for talking, and also listening to other participants
Participant.prototype = {
    send: function(message, to) {
        this.chatroom.send(message, this, to);
    },
    receive: function(message, from) {
        log.add(from.name + " to " + this.name + ": " + message);
    }
};

 // chatroom is the Mediator: it is the hub where participants send messages to, and receive messages
from
var Chatroom = function() {
    var participants = {};

    return {

        register: function(participant) {
            participants[participant.name] = participant;
            participant.chatroom = this;
        },

        send: function(message, from) {
            for (key in participants) {
                if (participants[key] !== from) {//you can't message yourself !
                    participants[key].receive(message, from);
                }
            }
        }

    };
};

// log helper

var log = (function() {
    var log = "";

    return {
        add: function(msg) { log += msg + "\n"; },
        show: function() { alert(log); log = ""; }
    }
```

```
})();

function run() {
    var yoko = new Participant("Yoko");
    var john = new Participant("John");
    var paul = new Participant("Paul");
    var ringo = new Participant("Ringo");

    var chatroom = new Chatroom();
    chatroom.register(yoko);
    chatroom.register(john);
    chatroom.register(paul);
    chatroom.register(ringo);

    yoko.send("All you need is love.");
    yoko.send("I love you John.");
    paul.send("Ha, I heard that!");

    log.show();
}
```

Section 79.3: Command

The command pattern encapsulates parameters to a method, current object state, and which method to call. It is useful to compartmentalize everything needed to call a method at a later time. It can be used to issue a "command" and decide later which piece of code to use to execute the command.

There are three components in this pattern:

1. Command Message - the command itself, including the method name, parameters, and state
2. Invoker - the part which instructs the command to execute its instructions. It can be a timed event, user interaction, a step in a process, callback, or any way needed to execute the command.
3. Receiver - the target of the command execution.

Command Message as an Array

```
var aCommand = new Array();
aCommand.push(new Instructions().DoThis);  //Method to execute
aCommand.push("String Argument");  //string argument
aCommand.push(777);                //integer argument
aCommand.push(new Object {} );     //object argument
aCommand.push(new Array() );       //array argument
```

Constructor for command class

```
class DoThis {
    constructor( stringArg, numArg, objectArg, arrayArg ) {
        this._stringArg = stringArg;
        this._numArg = numArg;
        this._objectArg = objectArg;
        this._arrayArg = arrayArg;
    }
    Execute() {
       var receiver = new Instructions();
       receiver.DoThis(this._stringArg, this._numArg, this._objectArg, this._arrayArg );
    }
}
```

Invoker

```
aCommand.Execute();
```

Can invoke:

- immediately
- in response to an event
- in a sequence of execution
- as a callback response or in a promise
- at the end of an event loop
- in any other needed way to invoke a method

Receiver

```
class Instructions {
    DoThis( stringArg, numArg, objectArg, arrayArg ) {
        console.log( `${stringArg}, ${numArg}, ${objectArg}, ${arrayArg}` );
    }
}
```

A client generates a command, passes it to an invoker that either executes it immediately or delays the command, and then the command acts upon a receiver. The command pattern is very useful when used with companion patterns to create messaging patterns.

Section 79.4: Iterator

An iterator pattern provides a simple method for selecting, sequentially, the next item in a collection.

Fixed Collection

```
class BeverageForPizza {
    constructor(preferenceRank) {
        this.beverageList = beverageList;
        this.pointer = 0;
    }
    next() {
        return this.beverageList[this.pointer++];
    }

var withPepperoni = new BeverageForPizza(["Cola", "Water", "Beer"]);
withPepperoni.next(); //Cola
withPepperoni.next(); //Water
withPepperoni.next(); //Beer
```

In ECMAScript 2015 iterators are a built-in as a method that returns done and value. done is true when the iterator is at the end of the collection

```
function preferredBeverage(beverage){
    if( beverage == "Beer" ){
        return true;
    } else {
        return false;
    }
}
var withPepperoni = new BeverageForPizza(["Cola", "Water", "Beer", "Orange Juice"]);
for( var bevToOrder of withPepperoni ){
```

```
    if( preferredBeverage( bevToOrder ) {
        bevToOrder.done; //false, because "Beer" isn't the final collection item
        return bevToOrder; //"Beer"
    }
}
```

As a Generator

```
class FibonacciIterator {
    constructor() {
        this.previous = 1;
        this.beforePrevious = 1;
    }
    next() {
        var current = this.previous + this.beforePrevious;
        this.beforePrevious = this.previous;
        this.previous = current;
        return current;
    }
}

var fib = new FibonacciIterator();
fib.next(); //2
fib.next(); //3
fib.next(); //5
```

In ECMAScript 2015

```
function* FibonacciGenerator() {  //asterisk informs javascript of generator
    var previous = 1;
    var beforePrevious = 1;
    while(true) {
        var current = previous + beforePrevious;
        beforePrevious = previous;
        previous = current;
        yield current;  //This is like return but
                        //keeps the current state of the function
                        // i.e it remembers its place between calls
    }
}

var fib = FibonacciGenerator();
fib.next().value; //2
fib.next().value; //3
fib.next().value; //5
fib.next().done; //false
```