Classic JavaScript Patterns

Gang of Four

Companion document to:

JavaScript + jQuery Design Pattern

Framework™ 2013

Index

Introduction

We refer to the Gang-of-Four patterns as the Classic patterns. They were first published in 1995 which is interesting because this was the same year that JavaScript was first released. However, the two (GoF and JavaScript) did not see eye to eye for about a decade. Only when JavaScript became more popular and widely used (in particular when Web 2.0 and Ajax were all the rage) started there to be an interest in GoF and patterns in general in the JavaScript world.

The Gang of Four (GoF) patterns are generally considered the foundation for all other patterns. Many of these are relevant and applicable to JavaScript, but others less so. Why are some not relevant? It is because JavaScript is a highly flexible language which allows runtime changes of objects, including the addition, removal, and change of the object's properties and methods. This flexibility is essentially what some classic patterns are aiming for, but since this is already built into the language itself, those patterns are not relevant.

The GoF listed a total of 23 patterns. They are categorized in three groups: Creational, Structural, and Behavioral. Here is the complete list.

Creational Patterns	
Abstract Factory	Creates an instance of several families of classes
Builder	Separates object construction from its representation
Factory Method	Creates an instance of several derived classes
Prototype	A fully initialized instance to be copied or cloned
Singleton	A class of which only a single instance can exist

Structural Patterns	
Adapter	Match interfaces of different classes
Bridge	Separates an object's interface from its implementation
Composite	A tree structure of simple and composite objects
Decorator	Add responsibilities to objects dynamically
Façade	A single class that represents an entire subsystem
Flyweight	A fine-grained instance used for efficient sharing
Proxy	An object representing another object

Behavioral Patterns	
Chain of Resp.	A way of passing a request between a chain of objects
Command	Encapsulate a command request as an object
Interpreter	A way to include language elements in a program
Iterator	Sequentially access the elements of a collection
Mediator	Defines simplified communication between classes

Memento	Capture and restore and object's internal state
Observer	A way of notifying change to a number of classes
State	Alter an object's behavior when its state changes
Strategy	Encapsulates an algorithm inside a class
Template Method	Defer the exact steps of an algorithm to a subclass
Visitor	Defines a new operation to a class without change

In this section, we review each of the 23 GoF patterns. Each pattern includes a description including a diagram (semi-UML), a traditional code implementation, and a JavaScript optimized code implementation. Let's get started with the first pattern which is Abstract Factory.

Abstract Factory

An Abstract Factory creates objects that are related by a common theme. In object-oriented programming a Factory is an object that creates other objects. An Abstract Factory has abstracted out a theme which is shared by the newly created objects.

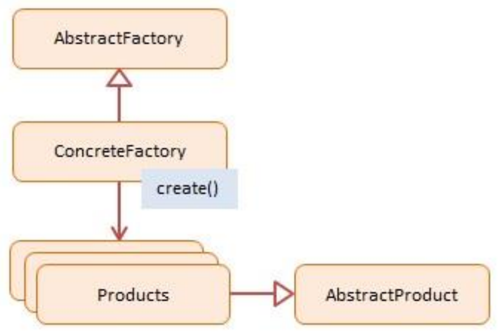
Suppose we have two Abstract Factories whose task it is to create page controls, such as, buttons, textboxes, radio buttons, and listboxes. One is the Light Factory which creates controls that are white and the other the Dark Factory which creates controls that are black. Both Factories creates the same types of controls, but they differ in color, which is their common theme. This is an implementation of the Abstract Factory pattern.

Over time the Abstract Factory and Factory Method patterns have merged into a more general pattern called Factory. A Factory is simply an object that creates other objects.

You may be wondering why you would want to leave the responsibility of the construction of objects to others rather than simply calling a constructor function with the new keyword directly. The reason is that that constructor functions are limited in their control over the overall creation process and sometimes you will need to hand over control to a factory that has broader knowledge.

This includes scenarios in which the creation process involves object caching, sharing or re-using of objects, complex logic, or applications that maintain object and type counts, and objects that interact with different resources or devices. If your application needs more control over the object creation process, consider using a Factory.

Diagram



Participants

The objects participating in this pattern are:

- AbstractFactory – not used in JavaScript
    - Declares an interface for creating products
- ConcreteFactory – in sample code: EmployeeFactory, VendorFactory
    - A factory object that manufactures new products
    - The create() method returns new products
- Products – In sample code: Employee, Vendor
    - The product instances being created by the factory
- Abstract Product – not used in JavaScript
    - Declares an interface for the products that are being created

JavaScript code

JavaScript does not support class-based inheritance therefore the abstract classes as depicted in the diagram are not used in the JavaScript sample. Abstract classes and interfaces enforce consistent interfaces in derived classes. In JavaScript we must ensure this consistency ourselves by making sure that each 'Concrete' object has the same interface definition (i.e. properties and methods) as the others.

In the example we have two Concrete Factories: `EmployeeFactory` and `VendorFactory`. The first one creates `Employee` instances, the second one `Vendor` instances. Both products are person types (with the same interface) which allow the client to treat them the same. An array with two employees and two vendors is created. Each person is then asked to say what and who they are.

The `log` function is a helper which collects and displays results.

```
function Employee(name) {
    this.name = name;
    this.say = function () {
        log.add("I am employee " + name);
    };
}
```

```
function EmployeeFactory() {
    this.create = function(name) {
        return new Employee(name);
    };
}

function Vendor(name) {
    this.name = name;
    this.say = function () {
        log.add("I am vendor " + name);
    };
}

function VendorFactory() {
    this.create = function(name) {
        return new Vendor(name);
    };
}

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


function run() {

    var persons = [];

    var employeeFactory = new EmployeeFactory();
    var vendorFactory = new VendorFactory();

    persons.push(employeeFactory.create("Joan DiSilva"));
    persons.push(employeeFactory.create("Tim O'Neill"));

    persons.push(vendorFactory.create("Gerald Watson"));
    persons.push(vendorFactory.create("Nicole McNight"));

    for (var i = 0, len = persons.length; i < len; i++) {
        persons[i].say();
    }

    log.show();
}
```

JavaScript Optimized Code

The Namespace pattern has been applied to keep the code out of the global namespace. Our namespace is named `Patterns.Classic`. A Revealing Module named `AbstractFactory` encapsulates all of Abstract Factory's functions. Only the `EmployeeFactory` and `VendorFactory` are exposed (revealed), whereas the `Employee` and `Vendor` constructor functions are kept private inside the module.

The `Patterns` object contains the `namespace` function which constructs namespaces non-destructively, that is, if a name already exists it won't overwrite it.

The `log` function is a helper which collects and displays results.

```
var Patterns = {
    namespace: function (name) {
        var parts = name.split(".");
        var ns = this;

        for (var i = 0, len = parts.length; i < len; i++) {
            ns[parts[i]] = ns[parts[i]] || {};
            ns = ns[parts[i]];
        }

        return ns;
    }
};

Patterns.namespace("Classic").AbstractFactory = (function () {

    var Employee = function (name) {
        this.name = name;
        this.say = function () {
            log.add("I am employee " + name);
        };
    };

    var Vendor = function (name) {
        this.name = name;
        this.say = function () {
            log.add("I am vendor " + name);
        };
    };

    var EmployeeFactory = function () {
        this.create = function (name) {
            return new Employee(name);
        };
    };

    var VendorFactory = function () {
        this.create = function (name) {
            return new Vendor(name);
        };
    };

    return {
        EmployeeFactory: EmployeeFactory,
        VendorFactory: VendorFactory
    };
})();

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


function run() {

    var abstract = Patterns.Classic.AbstractFactory;

    var employeeFactory = new abstract.EmployeeFactory();
    var vendorFactory = new abstract.VendorFactory();

    var persons = [];

    persons.push(employeeFactory.create("Joan DiSilva"));
    persons.push(employeeFactory.create("Tim O'Neill"));

    persons.push(vendorFactory.create("Gerald Watson"));
    persons.push(vendorFactory.create("Nicole McNight"));

    for (var i = 0, len = persons.length; i < len; i++) {
        persons[i].say();
    }

    log.show();
}
```

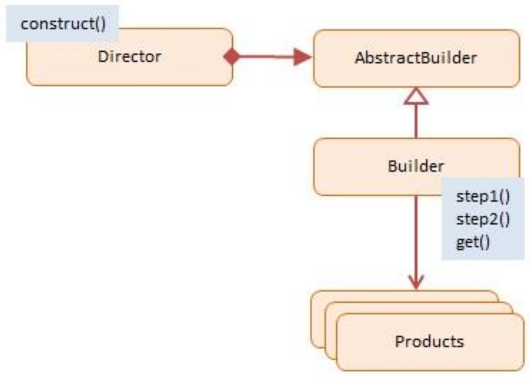
Builder

The Builder pattern allows a client to construct a complex object by specifying the type and content only. Construction details are hidden from the client entirely.

The most common motivation for using Builder is to simplify client code that creates complex objects. The client can still direct the steps taken by the Builder without knowing how the actual work is accomplished. Builders frequently encapsulate construction of Composite objects (another GoF design pattern) because the procedures involved are often repetitive and complex.

Usually it is the last step that returns the newly created object which makes it easy for a Builder to participate in fluent interfaces in which multiple method calls, separated by dot operators, are chained together (note: fluent interfaces are implementation of the Chaining Pattern as presented in the Modern patterns section).

Diagram



Participants

The objects participating in this pattern are:

- Director – in sample code: Shop
    - Constructs products by using the Builder's multistep interface
- Builder – not used in JavaScript
    - Declares a multistep interface for creating a complex object
- ConcreteBuilder – in sample code: CarBuilder, TruckBuilder
    - Implements the multistep Builder interface
    - Maintains the product through the assembly process
    - Offers the ability to retrieve the newly created product
- Products – in sample code: Car, Truck
    - Represents the complex objects being assembled

JavaScript Code

The AbstractBuilder is not used because JavaScript does not support abstract classes. However, the different Builders must implement the same multistep interface for the Director to be able to step through the assembly process.

The JavaScript code has a `Shop` (the Director) and two builder objects: `CarBuilder` and `TruckBuilder`. The Shop's `construct` method accepts a Builder instance which it then takes through a series of assembly steps: `step1` and `step2`. The Builder's `get` method returns the newly assembled products (`Car` objects and `Truck` objects).

The client has control over the actual object construction process by offering different builders to the `Shop`.

The `log` function is a helper which collects and displays results.

```
function Shop() {
    this.construct = function(builder) {
        builder.step1();
        builder.step2();
        return builder.get();
    }
}

function CarBuilder() {
    this.car = null;
    this.step1 = function() {
        this.car = new Car();
    };
    this.step2 = function() {
        this.car.addParts();
    };
    this.get = function() {
        return this.car;
    };
}

function TruckBuilder() {
    this.truck = null;
    this.step1 = function() {
        this.truck = new Truck();
    };
    this.step2 = function() {
        this.truck.addParts();
    };
    this.get = function() {
        return this.truck;
    };
}

function Car() {
    this.doors = 0;
    this.addParts = function() {
        this.doors = 4;
    };
    this.say = function() {
        log.add("I am a " + this.doors + "-door car");
    };
}

function Truck() {
    this.doors = 0;
    this.addParts = function() {
        this.doors = 2;
    };
    this.say = function() {
        log.add("I am a " + this.doors + "-door truck");
    };
}

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

function run() {
    var shop = new Shop();

    var carBuilder = new CarBuilder();
    var truckBuilder = new TruckBuilder();

    var car = shop.construct(carBuilder);
    var truck = shop.construct(truckBuilder);

    car.say();
    truck.say();

    log.show();
}
```

JavaScript Optimized Code

The Namespace pattern is applied to keep the code out of the global namespace. Our namespace is named `Patterns.Classic`. A Revealing Module named `Builder` encapsulates all of Builder's functions. Only the `Shop`, `CarBuilder`, and `TruckBuilder` are exposed (revealed), whereas the `Car` and `Truck` constructor functions are kept private inside the module.

The `Patterns` object contains the `namespace` function which constructs namespaces non-destructively, that is, if a name already exists it won't overwrite it.

The `log` function is a helper which collects and displays results.

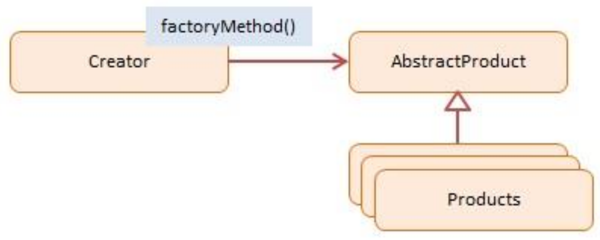
```
var Patterns = {
    namespace: function (name) {
        var parts = name.split(".");
        var ns = this;

        for (var i = 0, len = parts.length; i < len; i++) {
            ns[parts[i]] = ns[parts[i]] || {};
            ns = ns[parts[i]];
        }
        return ns;
    }
};

Patterns.namespace("Classic").Builder = (function () {

    var Car = function () {
        this.doors = 0;
        this.addParts = function () {
            this.doors = 4;
        };
        this.say = function () {
            log.add("I am a " + this.doors + "-door car");
        };
    };

    var Truck = function () {
        this.doors = 0;
        this.addParts = function () {
            this.doors = 2;
        };
        this.say = function () {
            log.add("I am a " + this.doors + "-door truck");
        };
    };

    var Shop = function () {
        this.construct = function (builder) {
            builder.step1();
            builder.step2();
            return builder.get();
        }
    };

    var CarBuilder = function () {
        this.car = null;
        this.step1 = function () {
            this.car = new Car();
        };
        this.step2 = function () {
            this.car.addParts();
        };
        this.get = function () {
            return this.car;
        }
    };

    var TruckBuilder = function () {
        this.truck = null;
        this.step1 = function () {
            this.truck = new Truck();
        };
        this.step2 = function () {
            this.truck.addParts();
        };
        this.get = function () {
            return this.truck;
        };
    };

    return {
        Shop: Shop,
        CarBuilder: CarBuilder,
        TruckBuilder: TruckBuilder
    };
})();


function run() {

    var builder = Patterns.Classic.Builder;

    var carBuilder = new builder.CarBuilder();
    var truckBuilder = new builder.TruckBuilder();

    var shop = new builder.Shop();

    var car = shop.construct(carBuilder);
    var truck = shop.construct(truckBuilder);

    car.say();
    truck.say();

    log.show();
}
```

Factory Method

A Factory Method creates new objects as instructed by the client. One way to create objects in JavaScript is by invoking a constructor function with the new operator. There are situations however, where the client does not, or should not, know which one of several candidate objects to instantiate. The Factory Method allows the client to delegate object creation while still retaining control over which type to instantiate.

The key objective of the Factory Method is extensibility. Factory Methods are frequently used in applications that manage, maintain, or manipulate collections of objects that are different but at the same time have many characteristics (i.e. methods and properties) in common. An example would be a collection of documents with a mix of Xml documents, Pdf documents, and Rtf documents.

Diagram



Participants

The objects participating in this pattern are:

- Creator -- In sample code: Factory
    - the 'factory' object that creates new products
    - implements 'factoryMethod' which returns newly created products
- AbstractProduct -- not used in JavaScript
    - declares an interface for products
- ConcreteProduct -- In sample code: Employees
    - the product being created
    - all products support the same interface (properties and methods)

JavaScript Code

In this JavaScript example the `Factory` object creates four different types of employees. Each employee type has a different hourly rate. The `createEmployee` method is the actual Factory Method. The client instructs the factory what type of employee to create by passing a type argument into the Factory Method.

The AbstractProduct in the diagram is not implemented because JavaScript does not support abstract classes or interfaces. However, we still need to ensure that all employee types have the same interface (properties and methods).

Four different employee types are created; all are stored in the same array. Each employee is asked to say what they are and their hourly rate.

The `log` function is a helper which collects and displays results.

```
function Factory() {
    this.createEmployee = function (type) {
        var employee;

        if (type === "fulltime") {
            employee = new FullTime();
        } else if (type === "parttime") {
            employee = new PartTime();
        } else if (type === "temporary") {
            employee = new Temporary();
        } else if (type === "contractor") {
            employee = new Contractor();
        }

        employee.type = type;
        employee.say = function () {
            log.add(this.type + ": rate " + this.hourly + "/hour");
        }
        return employee;
    }
}

var FullTime = function () {
    this.hourly = "$12";
};
var PartTime = function () {
    this.hourly = "$11";
};
var Temporary = function () {
    this.hourly = "$10";
};
var Contractor = function () {
    this.hourly = "$15";
};

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


function run() {

    var employees = [];

    var factory = new Factory();

    employees.push(factory.createEmployee("fulltime"));
    employees.push(factory.createEmployee("parttime"));
    employees.push(factory.createEmployee("temporary"));
    employees.push(factory.createEmployee("contractor"));

    for (var i = 0, len = employees.length; i < len; i++) {
        employees[i].say();
    }

    log.show();
}
```

JavaScript Optimized Code

The Namespace pattern is applied to keep the code out of the global namespace. Our namespace is named `Patterns.Classic`. A Revealing Module named `Factory` holds all of Factory Method's functions. Only the actual Factory Method named `createEmployee` is exposed (revealed), whereas all other functions `FullTime`, `PartTime`, `Temporary`, and `Contractor` constructor functions are kept private inside the module.

The `Patterns` object contains the `namespace` function which constructs namespaces non-destructively, that is, if a name already exists it won't overwrite it.

The `log` function is a helper which collects and displays results.

```
var Patterns = {
    namespace: function (name) {
        var parts = name.split(".");
        var ns = this;

        for (var i = 0, len = parts.length; i < len; i++) {
            ns[parts[i]] = ns[parts[i]] || {};
            ns = ns[parts[i]];
        }

        return ns;
    }
};

Patterns.namespace("Classic").Factory = (function () {

    var createEmployee = function (type) {
        var employee;

        if (type === "fulltime") {
            employee = new FullTime();
        } else if (type === "parttime") {
            employee = new PartTime();
        } else if (type === "temporary") {
            employee = new Temporary();
        } else if (type === "contractor") {
            employee = new Contractor();
        }

        employee.type = type;
        employee.say = function () {
            log.add(this.type + ": rate " + this.hourly + "/hour");
        }
        return employee;
    };

    var FullTime = function () {
        this.hourly = "$12";
    };
    var PartTime = function () {
        this.hourly = "$11";
    };
    var Temporary = function () {
        this.hourly = "$10";
    };
    var Contractor = function () {
        this.hourly = "$15";
    };

    return {
        createEmployee: createEmployee
    };

})();

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


function run() {

    var employees = [];

    var factory = Patterns.Classic.Factory;

    employees.push(factory.createEmployee("fulltime"));
    employees.push(factory.createEmployee("parttime"));
    employees.push(factory.createEmployee("temporary"));
    employees.push(factory.createEmployee("contractor"));

    for (var i = 0, len = employees.length; i < len; i++) {
        employees[i].say();
    }

    log.show();
}
```

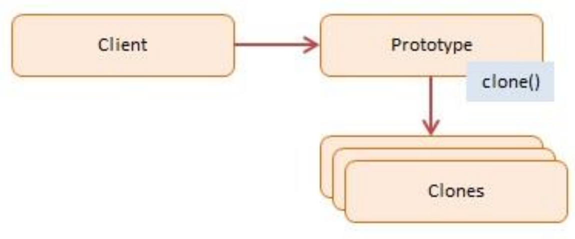
Prototype

The Prototype Pattern creates new objects, but rather than creating non-initialized objects it returns objects that are initialized with values it copied from a prototype - or sample - object. The Prototype pattern is also referred to as the Properties pattern.

An example of where the Prototype pattern is useful is the initialization of business objects with values that match the default values in the database. The prototype object holds the default values that are copied over into a newly created business object.

Classical languages rarely use the Prototype pattern, but JavaScript being a prototypal language uses this pattern in the construction of new objects and their prototypes.

Diagram



Participants

The objects participating in this pattern are:

- Client -- In sample code: the run() function
    - creates a new object by asking a prototype to clone itself
- Prototype -- In sample code: CustomerPrototype
    - creates an interfaces to clone itself
- Clones -- In sample code: Customer
    - the cloned objects that are being created

JavaScript Code

In the sample code we have a `CustomerPrototype` object which clones objects given a prototype object. Its constructor function accepts a prototype of type `Customer`. Calling the `clone` method generates a new `Customer` object with its property values initialized with the prototype values.

This is the classical implementation of the Prototype pattern, but JavaScript can do this far more effectively using its built-in prototype facility. We will explore this in the JavaScript optimized code.

```
function CustomerPrototype(proto) {
    this.proto = proto;

    this.clone = function () {
        var customer = new Customer();

        customer.first = proto.first;
        customer.last = proto.last;
        customer.status = proto.status;

        return customer;
    };
}

function Customer(first, last, status) {

    this.first = first;
    this.last = last;
    this.status = status;

    this.say = function () {
        alert("name: " + this.first + " " + this.last +
              ", status: " + this.status);
    };
}


function run() {

    var proto = new Customer("n/a", "n/a", "pending");
    var prototype = new CustomerPrototype(proto);

    var customer = prototype.clone();
    customer.say();
}
```

JavaScript Optimized Code

The Namespace pattern is applied to keep the code out of the global namespace. Our namespace is named `Patterns.Classic`. A Revealing Module named Prototype holds all of Prototype's functions. It exposes the `Customer` constructor function and by association its prototype.

In the run function, first a default customer is created in which all properties have default values. Then a second customer is created by providing two property values: Kevin and Summer as first and last name respectively.

Notice that when overriding the defaults, we are not changing the prototype values. Instead, two new properties are added to the Customer object itself: first and last. These new property values hide the prototype values.

The `Patterns` object contains the namespace function which constructs namespaces non-destructively, that is, if a name already exists it won't overwrite it.

The `log` function is a helper which collects and displays results.

```
var Patterns = {
    namespace: function (name) {
        var parts = name.split(".");
        var ns = this;

        for (var i = 0, len = parts.length; i < len; i++) {
            ns[parts[i]] = ns[parts[i]] || {};
            ns = ns[parts[i]];
        }

        return ns;
    }
};


Patterns.namespace("Classic").Prototype = (function () {

    function Customer(first, last, status) {

        if (first) this.first = first;
        if (last) this.last = last;
        if (status) this.status = status;
    }

    Customer.prototype = {
        say: function () {
            log.add("name: " + this.first + " " + this.last +
                    ", status: " + this.status);
        },
        first: "n/a",
        last: "n/a",
        status: "pending"
    };

    return { Customer: Customer };

})();

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


function run() {
    var proto = Patterns.Classic.Prototype;

    var customer = new proto.Customer();
    customer.say();

    var kevin = new proto.Customer("Kevin", "Summer");
    kevin.say();

    log.show();
}
```

Singleton

The Singleton Pattern limits the number of instances of a particular object to just one. This single instance is called the singleton.

Singletons are useful in situations where system-wide actions need to be coordinated from a single central place. An example is a database connection pool. The pool manages the creation, destruction, and lifetime of all database connections for the entire application ensuring that no connections are 'lost'.

Singletons reduce the need for global variables which is particularly important in JavaScript because it limits namespace pollution and associated risk of name collisions. The Module pattern (described in the Modern Patterns section) is JavaScript's manifestation of the Singleton pattern.

Several other patterns, such as, Factory, Prototype, and Façade are frequently implemented as Singletons when only one instance is needed.

Diagram

getInstance()
Singleton

Participants

The objects participating in this pattern are:

- Singleton -- In sample code: Singleton
  - defines getInstance() which returns the unique instance.
  - responsible for creating and managing the instance object.

JavaScript Code

The `Singleton` object is implemented as an immediate anonymous function. The function executes immediately by wrapping it in brackets followed by two additional brackets. It is called anonymous because it doesn't have a name.

The `getInstance` method is Singleton's gatekeeper method. It returns the one and only instance of the object while maintaining a private reference to it which is not accessible to the outside world.

The `getInstance` method demonstrates another design pattern called Lazy Load. Lazy Load checks if an instance has already been created; if not, it creates one and stores it for future reference. All subsequent calls will receive the stored instance. Lazy loading is a CPU and memory saving technique by creating objects only when absolutely necessary.

Singleton is a manifestation of a common JavaScript pattern: the Module pattern. Module is the basis to all popular JavaScript libraries and frameworks (jQuery, Backbone, Ember, etc.). The Module and Lazy Load patterns are discussed in the Modern Patterns section.

```
var Singleton = (function () {

    var instance;

    function createInstance() {
        var object = new Object("I am the instance");
        return object;
    }

    return {
        getInstance: function () {
            if (!instance) {
                instance = createInstance();
            }

            return instance;
        }
    };
})();


function run() {

    var instance1 = Singleton.getInstance();
    var instance2 = Singleton.getInstance();

    alert("Same instance? " + (instance1 === instance2));
}
```

JavaScript Optimized Code

The Namespace pattern is applied to keep the code out of the global namespace. Our namespace is named `Patterns.Classic`. A Revealing Module named `Singleton` holds all of Singleton's functions and variables. It exposes just one method: `getInstance`. This method will return the one and only instance if it already exists; if not it will create a new instance and save it off in a private local variable for all subsequent calls.

The `Patterns` object contains the `namespace` function which constructs namespaces non-destructively, that is, if a name already exists it won't overwrite it.

The `log` function is a helper which collects and displays results.

```
var Patterns = {
    namespace: function (name) {
        var parts = name.split(".");
        var ns = this;

        for (var i = 0, len = parts.length; i < len; i++) {
            ns[parts[i]] = ns[parts[i]] || {};
            ns = ns[parts[i]];
        }

        return ns;
    }
};

Patterns.namespace("Classic").Singleton = (function () {

    var instance;

    var createInstance = function () {
        return {
            say: function () {
                alert("I am the greatest");
            }
        };
    }

    var getInstance = function () {
        return instance = instance || createInstance();
    }

    return {
        getInstance: getInstance
    };

})();


function run() {

    var singleton = Patterns.Classic.Singleton;

    var instance1 = singleton.getInstance();
    var instance2 = singleton.getInstance();

    instance1.say();                          // => I am the greatest!

    alert("Same instance? " + (instance1 === instance2)); // => true
}
```

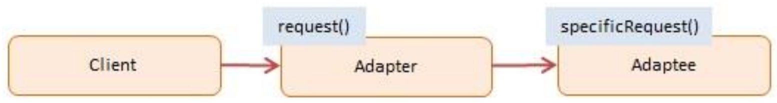
Adapter

The Adapter pattern translates one interface (an object's properties and methods) to another. Adapters allows programming components to work together that otherwise wouldn't because of mismatched interfaces. The Adapter pattern is also referred to as the Wrapper Pattern.

One scenario where Adapters are commonly used is when new components need to be integrated and work together with existing components in the application.

Another scenario is refactoring in which parts of the program are rewritten with an improved interface, but the old code still expects the original interface.

Diagram



Participants

The objects participating in this pattern are:

- Client -- In sample code: the run() function.
  - calls into Adapter to request a service
- Adapter -- In sample code: ShippingAdapter
  - implements the interface that the client expects or knows
- Adaptee -- In sample code: AdvancedShipping
  - the object being adapted
  - has a different interface from what the client expects or knows

JavaScript Code

The example code below shows an online shopping cart in which a shipping object is used to compute shipping costs. The old `Shipping` object is replaced by a new and improved Shipping object that is more secure and offers better prices.

This new object is named `AdvancedShipping` and has a very different interface which the client program does not expect. `ShippingAdapter` allows the client program to continue functioning without any API changes by mapping (adapting) the old `Shipping` interface to the new `AdvancedShipping` interface.

The `log` function is a helper which collects and displays results.

```
// old interface
function Shipping() {
    this.request = function(zipStart, zipEnd, weight) {
        // ...
        return "$49.75";
    }
}

// new interface
function AdvancedShipping() {
    this.login = function(credentials) { /* ... */ };
    this.setStart = function(start) { /* ... */ };
    this.setDestination = function(destination) { /* ... */ };
    this.calculate = function(weight) { return "$39.50"; };
}

// adapter interface
function ShippingAdapter(credentials) {
    var shipping = new AdvancedShipping();
    shipping.login(credentials);

    return {
        request: function(zipStart, zipEnd, weight) {
            shipping.setStart(zipStart);
            shipping.setDestination(zipEnd);
            return shipping.calculate(weight);
        }
    };
}

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


function run() {

    var shipping = new Shipping();

    var credentials = {token: "30a8-6ee1"};
    var adapter = new ShippingAdapter(credentials);

    // original shipping object and interface
    var cost = shipping.request("78701", "10010", "2 lbs");
    log.add("Old cost: " + cost);

    // new shipping object with adapted interface
    cost = adapter.request("78701", "10010", "2 lbs");
    log.add("New cost: " + cost);

    log.show();
}
```

JavaScript Optimized Code

The Namespace pattern is applied to keep the code out of the global namespace. Our namespace is named `Patterns.Classic`. A Revealing Module named `Adapter` holds all of Adapter's functions including the advanced shipping interface. It only exposes `ShippingAdapter` which has a single request method similar to the old shipping interface.

The original Shipping interface was kept outside our namespace because it is only used to show that the new (adapted) and old API are the same.

The `Patterns` object contains the `namespace` function which constructs namespaces non-destructively, that is, if a name already exists it won't overwrite it.

The `log` function is a helper which collects and displays results.

```
var Patterns = {
    namespace: function (name) {
        var parts = name.split(".");
        var ns = this;

        for (var i = 0, len = parts.length; i < len; i++) {
            ns[parts[i]] = ns[parts[i]] || {};
            ns = ns[parts[i]];
        }

        return ns;
    }
};

Patterns.namespace("Classic").Adapter = (function () {

    // new interface
    var AdvancedShipping = function () {
        this.login = function (credentials) { /* ... */ };
        this.setStart = function (start) { /* ... */ };
        this.setDestination = function (destination) { /* ... */ };
        this.calculate = function (weight) { return "$39.50"; };
    }

    // adapter interface
    var ShippingAdapter = function (credentials) {
        var shipping = new AdvancedShipping();
        shipping.login(credentials);

        return {
            request: function (zipStart, zipEnd, weight) {
                shipping.setStart(zipStart);
                shipping.setDestination(zipEnd);
                return shipping.calculate(weight);
            }
        };
    };
    return {
        ShippingAdapter: ShippingAdapter
    };

})();

// old interface
var Shipping = function () {
    this.request = function (zipStart, zipEnd, weight) {
        // ...
        return "$49.75";
    }
}

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


function run() {

    var shipping = new Shipping();

    var credentials = { token: "30a8-6ee1" };
    var adapter = new Patterns.Classic.Adapter.ShippingAdapter(credentials);

    // original shipping object and interface
    var cost = shipping.request("78701", "10010", "2 lbs");
    log.add("Old cost: " + cost);

    // new shipping object with adapted interface
    cost = adapter.request("78701", "10010", "2 lbs");
    log.add("New cost: " + cost);

    log.show();
}
```

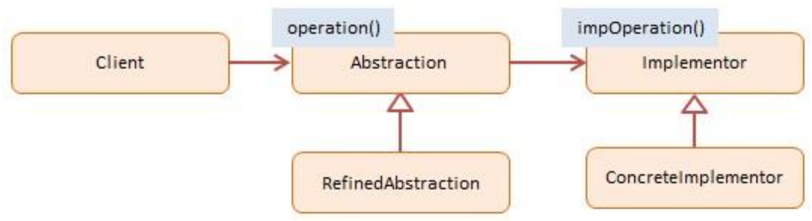
Bridge

The Bridge pattern allows two components, a client and a service, to work together with each component having its own interface. Bridge is a high-level architectural pattern and its main goal is to write better code through two levels of abstraction. It facilitates very loose coupling of objects. It is sometimes referred to as a double Adapter pattern.

An example of the Bridge pattern is an application (the client) and a database driver (the service). The application writes to a well-defined database API, for example ODBC, but behind this API you will find that each driver's implementation is totally different for each database vendor (SQL Server, MySQL, Oracle, etc.).

The Bridge pattern is a great pattern for driver development but it is rarely seen in JavaScript.

Diagram



Participants

The objects participating in this pattern are:

- Client -- In sample code: the run() function.
    - calls into Abstraction to request an operation
- Abstraction -- not used in JavaScript
    - declares an interface for first level abstraction
    - maintains a reference to the Implementor
- RefinedAbstraction -- In sample code: Gestures, Mouse
    - implements and extends the interface defined by Abstraction
- Implementor -- not used in JavaScript
    - declares an interface for second level or implementor abstraction
- ConcreteImplementor -- In sample code: Screen, Audio
    - implements the Implementor interface and defines its effects

JavaScript Code

The objective of the example is to show that with the Bridge pattern input and output devices can vary independently (without changes to the code); the devices are loosely coupled by two levels of abstraction.

JavaScript does not support abstract classes therefore Abstraction and Implementor are not included. However, their interfaces (properties and methods) are consistently implemented in RefinedAbstraction and ConcreteImplementor. In our example code the Abstraction represents Input devices and the Implementor represents Output devices.

`Gestures` (finger movements) and the `Mouse` are very different input devices, but their actions map to a common set of output instructions: click, move, drag, etc. `Screen` and `Audio` are very different output devices, but they respond to the same set of instructions. Of course, the effects are totally different, that is, video updates vs. sound effects. The Bridge pattern allows any input device to work with any output device.

The `log` function is a helper which collects and displays results.

```
// input devices

var Gestures = function (output) {
    this.output = output;
    this.tap = function () { this.output.click(); }
    this.swipe = function () { this.output.move(); }
    this.pan = function () { this.output.drag(); }
    this.pinch = function () { this.output.zoom(); }
};

var Mouse = function (output) {
    this.output = output;
    this.click = function () { this.output.click(); }
    this.move = function () { this.output.move(); }
    this.down = function () { this.output.drag(); }
    this.wheel = function () { this.output.zoom(); }
};

// output devices

var Screen = function () {
    this.click = function () { log.add("Screen select"); }
    this.move = function () { log.add("Screen move"); }
    this.drag = function () { log.add("Screen drag"); }
    this.zoom = function () { log.add("Screen zoom in"); }
};

var Audio = function () {
    this.click = function () { log.add("Sound oink"); }
    this.move = function () { log.add("Sound waves"); }
    this.drag = function () { log.add("Sound screetch"); }
    this.zoom = function () { log.add("Sound volume up"); }
};

// logging helper

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


function run() {
    var screen = new Screen();
    var audio = new Audio();

    var hand = new Gestures(screen);
    var mouse = new Mouse(audio);

    hand.tap();
    hand.swipe();
    hand.pinch();

    mouse.click();
    mouse.move();
    mouse.wheel();

    log.show();
}
```

JavaScript Optimized Code

The Namespace pattern is applied to keep the code out of the global namespace. Our namespace is named `Patterns.Classic`. A Module named `Bridge` returns all functions that make up the Bridge pattern. This includes input devices, output devices and all their methods.

The `Patterns` object contains the `namespace` function which constructs namespaces non-destructively, that is, if a name already exists it won't overwrite it.

The `log` function is a helper which collects and displays results.

```
var Patterns = {
    namespace: function (name) {
        var parts = name.split(".");
        var ns = this;

        for (var i = 0, len = parts.length; i < len; i++) {
            ns[parts[i]] = ns[parts[i]] || {};
            ns = ns[parts[i]];
        }

        return ns;
    }
};

Patterns.namespace("Classic").Bridge = (function () {

    return {

        // input devices
        Gestures: function (output) {
            this.tap = function () { output.click(); }
            this.swipe = function () { output.move(); }
            this.pan = function () { output.drag(); }
            this.pinch = function () { output.zoom(); }
        },

        Mouse: function (output) {
            this.click = function () { output.click(); }
            this.move = function () { output.move(); }
            this.down = function () { output.drag(); }
            this.wheel = function () { output.zoom(); }
        },

        // output devices
        Screen: function () {
            this.click = function () { log.add("Screen select"); }
            this.move = function () { log.add("Screen move"); }
            this.drag = function () { log.add("Screen drag"); }
            this.zoom = function () { log.add("Screen zoom in"); }
        },

        Audio: function () {
            this.click = function () { log.add("Sound oink"); }
            this.move = function () { log.add("Sound waves"); }
            this.drag = function () { log.add("Sound screetch"); }
            this.zoom = function () { log.add("Sound volume up"); }
        }

    };

})();

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


function run() {

    var bridge = Patterns.Classic.Bridge;

    var screen = new bridge.Screen();
    var audio = new bridge.Audio();

    var hand = new bridge.Gestures(screen);
    var mouse = new bridge.Mouse(audio);

    hand.tap();
    hand.swipe();
    hand.pinch();

    mouse.click();
    mouse.move();
    mouse.wheel();

    log.show();
}
```

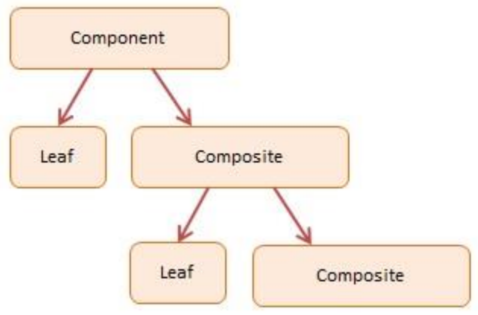
Composite

The Composite pattern allows the creation of objects with properties that are primitive items or a collection of objects. Each item in the collection can hold other collections themselves, creating deeply nested structures.

A tree control is a perfect example of a Composite pattern. The nodes of the tree either contain an individual object (leaf node) or a group of objects (a subtree of nodes).

All nodes in the Composite pattern share a common set of properties and methods which supports individual objects as well as object collections. This common interface greatly facilitates the design and construction of recursive algorithms that iterate over each object in the Composite collection.

Diagram



Participants

The objects participating in this pattern are:

- Component -- In sample code: Node
    - declares the interface for objects in the composition
- Leaf -- In sample code: Node
    - represents leaf objects in the composition. A leaf has no children
- Composite -- In sample code: Node
    - represents branches (or subtrees) in the composition
    - maintains a collection of child components

JavaScript Code

In our example a tree structure is created from `Node` objects. Each node has a name and 4 methods: `add`, `remove`, `getChild`, and `hasChildren`. These methods are placed on the `Node`'s prototype which reduces the memory requirements as these methods are now shared by all nodes. Node is fully recursive and there is no need for separate Component or Leaf objects.

A small Composite tree is built by adding nodes to parent nodes. Once complete we invoke `traverse` which iterates over each node in the tree and displays its name and depth (by showing indentation).

The `log` function is a helper which collects and displays results.

```
var Node = function (name) {
    this.children = [];
    this.name = name;
}

Node.prototype = {
    add: function (child) {
        this.children.push(child);
    },
    remove: function (child) {
        var len = this.children.length;
        for (var i = 0; i < len; i++) {
            if (this.children[i] === child) {
                this.children.splice(i, 1);
                return;
            }
        }
    },
    getChild: function (i) {
        return this.children[i];
    },
    hasChildren: function () {
        return this.children.length > 0;
    }
}

// recursively traverse a (sub)tree
function traverse(indent, node) {

    log.add(Array(indent++).join("--") + node.name);

    for (var i = 0, len = node.children.length; i < len; i++) {
        traverse(indent, node.getChild(i));
    }
}

// logging helper
var log = (function () {
    var log = "";
    return {
        add: function (msg) { log += msg + "\n"; },
        show: function () { alert(log); log = ""; }
    }
})();


function run() {

    var tree = new Node("root");
    var left = new Node("left")
    var right = new Node("right");
    var leftleft = new Node("leftleft");
    var leftright = new Node("leftright");
    var rightleft = new Node("rightleft");
    var rightright = new Node("rightright");

    tree.add(left);
    tree.add(right);
    tree.remove(right);  // note: remove
    tree.add(right);
    left.add(leftleft);
    left.add(leftright);
    right.add(rightleft);
    right.add(rightright);

    traverse(1, tree);

    log.show();
}
```

JavaScript Optimized Code

The Namespace pattern is applied to keep the code out of the global namespace. Our namespace is named `Patterns.Classic`. A Revealing Module named `Composite` returns all elements that make up the Composite pattern. They are a `Node` object and a recursive `traverse` method.

The `Patterns` object contains the `namespace` function which constructs namespaces non-destructively, that is, if a name already exists it won't overwrite it.

The `log` function is a helper which collects and displays results.

```
var Patterns = {
    namespace: function (name) {
        var parts = name.split(".");
        var ns = this;

        for (var i = 0, len = parts.length; i < len; i++) {
            ns[parts[i]] = ns[parts[i]] || {};
            ns = ns[parts[i]];
        }
        return ns;
    }
};

Patterns.namespace("Classic").Composite = (function () {

    var Node = function (name) {
        this.children = [];
        this.name = name;
    }

    Node.prototype = {
        add: function (child) {
            this.children.push(child);
        },
        remove: function (child) {
            var length = this.children.length;
            for (var i = 0; i < length; i++) {
                if (this.children[i] === child) {
                    this.children.splice(i, 1);
                    return;
                }
            }
        },
        getChild: function (i) {
            return this.children[i];
        },
        hasChildren: function () {
            return this.children.length > 0;
        }
    };

    // recursively traverse a (sub)tree
    var traverse = function (indent, node) {

        log.add(Array(indent++).join("--") + node.name);

        for (var i = 0, len = node.children.length; i < len; i++) {
            this.traverse(indent, node.getChild(i));
        }
    };

    return {
        Node: Node,
        traverse: traverse
    };

})();

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


function run() {

    var composite = Patterns.Classic.Composite;

    var tree = new composite.Node("root");
    var left = new composite.Node("left")
    var right = new composite.Node("right");
    var leftleft = new composite.Node("leftleft");
    var leftright = new composite.Node("leftright");
    var rightleft = new composite.Node("rightleft");
    var rightright = new composite.Node("rightright");

    tree.add(left);
    tree.add(right);
    tree.remove(right);  // note: remove
    tree.add(right);
    left.add(leftleft);
    left.add(leftright);
    right.add(rightleft);
    right.add(rightright);

    composite.traverse(1, tree);

    log.show();
}
```

Decorator

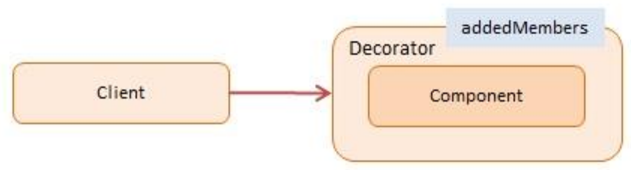
The Decorator pattern extends (decorates) an object's behavior dynamically. The ability to add new behavior at runtime is accomplished by a Decorator object which 'wraps itself' around the original object. Multiple decorators can add or override functionality to the original object.

An example of a decorator is security management where business objects are given additional access to privileged information depending on the privileges of the authenticated user. For example, an HR manager gets to work with an employee object that has appended (i.e. is decorated with) the employee's salary record so that salary information can be viewed.

Decorators provide flexibility to statically typed languages by allowing runtime changes as opposed to inheritance which takes place at compile time. JavaScript, however, is a dynamic language and the ability to extend an object at runtime is baked into the language itself.

For this reason, the Decorator pattern is less relevant to JavaScript developers. In JavaScript the Extend and Mixin patterns subsume the Decorator pattern. We will look at this in the JavaScript optimized code.

Diagram



Participants

The objects participating in this pattern are:

- Client -- In sample code: the run() function
    - maintains a reference to the decorated Component
- Component -- In sample code: User
    - object to which additional functionality is added
- Decorator -- In sample code: DecoratedUser
    - 'wraps around' Component by maintaining a reference to it
    - defines an interface that conforms to Component's interface
    - implements the additional functionality (addedMembers in diagram)

JavaScript Code

In the example code a `User` object is decorated (enhanced) by a `DecoratedUser` object. It extends the User with several address-based properties. The original interface must stay the same, which explains why `user.name` is assigned to `this.name`. Also, the `say` method of DecoratedUser hides the `say` method of User.

JavaScript itself is far more effective in dynamically extending objects with additional data and behavior. You can learn more about extending objects in the Modern Patterns under Mixin pattern. The JavaScript optimized solution below also demonstrates a better solution in JavaScript.

The `log` function is a helper which collects and displays results.

```
var User = function(name) {
    this.name = name;
    this.say = function() {
        log.add("User: " + this.name);
    };
}

var DecoratedUser = function(user, street, city) {
    this.user = user;
    this.name = user.name;  // ensures interface stays the same
    this.street = street;
    this.city = city;
    this.say = function() {
        log.add("Decorated User: " + this.name + ", " +
                    this.street + ", " + this.city);
    };
}

// logging helper
var log = (function() {
    var log = "";
    return {
        add: function(msg) { log += msg + "\n"; },
        show: function() { alert(log); log = ""; }
    }
})();


function run() {

    var user = new User("Kelly");
    user.say();

    var decorated = new DecoratedUser(user, "Broadway", "New York");
    decorated.say();

    log.show();
}
```

JavaScript Optimized Code

The Namespace pattern is applied to keep the code out of the global namespace. Our namespace is named `Patterns.Classic`. A Revealing Module named `Decorator` returns two methods: `extend` and `extendDeep`. They are described in the Modern Patterns section under the Mixin pattern (another 'decorator'). The extend method extends an object with additional properties and methods. The `extendDeep` method also extends an object but does it recursively which includes nested objects and arrays.

The `Patterns` object contains the `namespace` function which constructs namespaces non-destructively, that is, if a name already exists it won't overwrite it.

The `log` function is a helper which collects and displays results.

```
var Patterns = {
    namespace: function (name) {
        var parts = name.split(".");
        var ns = this;

        for (var i = 0, len = parts.length; i < len; i++) {
            ns[parts[i]] = ns[parts[i]] || {};
            ns = ns[parts[i]];
        }

        return ns;
    }
};

Patterns.namespace("Classic").Decorator = (function () {

    var extend = function (dest, source) {
        for (var prop in source) {
            if (source.hasOwnProperty(prop)) {
                dest[prop] = source[prop];
            }
        }
    };

    var extendDeep = function (dest, source) {
        for (var prop in source) {
            if (source.hasOwnProperty(prop)) {
                if (typeof prop === "object") {
                    dest[prop] = $.isArray(prop) ? [] : {};
                    this.deepExtend(dest[prop], source[prop]);
                }
                else {
                    dest[prop] = source[prop];
                }
            }
        }
    };

    return {
        extend: extend,
        extendDeep: extendDeep
    };

})();

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


function run() {

    var decorator = Patterns.Classic.Decorator;

    var User = function (name) {
        this.name = name;
        this.say = function () {
            log.add("User: " + this.name);
        };
    }
    var user = new User("Kelly");
    user.say();

    decorator.extend(user, {
        street: "Broadway",
        city: "New York",
        say: function () {
            log.add("Extended User: " + this.name + ", " +
                    this.street + ", " + this.city);
        }
    });
    user.say();

    decorator.extendDeep(user, {
        school: "Columbia",
        grades: {
            "Spring": 4.0,
            "Fall": 3.5
        },
        say: function () {
            log.add("Deeply Extended User: " + this.name + ", " +
                    this.street + ", " + this.city + ", " +
                    this.school + ", grades: " +
                    this.grades.Spring + ", " + this.grades.Fall);
        }
    });
    user.say();

    log.show();
}
```

Façade

The Façade pattern provides an interface which shields clients from complex functionality in one or more subsystems. It is a simple pattern that may seem trivial but it is powerful and extremely useful. It is often present in systems that are built around a multi-layer architecture.

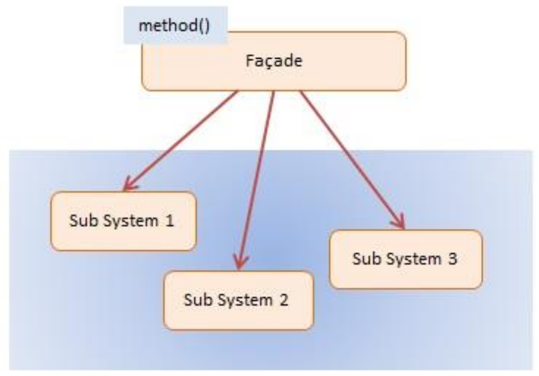
The intent of the Façade is to provide a high-level interface (properties and methods) that makes a subsystem or toolkit easy to use for the client.

On the server, in a multi-layer web application you frequently have a presentation layer which is a client to a service layer. Communication between these two layers takes place via a well-defined API. This API, or façade, hides the complexities of the business objects and their interactions from the presentation layer.

Another area where Façades are used is in refactoring. Suppose you have a confusing or messy set of legacy objects that the client should not be concerned about. You can hide this code behind a Façade. The Façade exposes only what is necessary and presents a cleaner and easy-to-use interface.

Façades are frequently combined with other design patterns. Facades themselves are often implemented as singleton factories.

Diagram



Participants

The objects participating in this pattern are:

- Facade -- In sample code: Mortgage
    - knows which subsystems are responsible for a request
    - delegates client requests to appropriate subsystem objects
- Sub Systems -- In sample code: Bank, Credit, Background
    - implements and performs specialized subsystem functionality
    - have no knowledge of or reference to the facade

JavaScript Code

The `Mortgage` object is the Facade in the sample code. It presents a simple interface to the client with only a single method: `applyFor`. But underneath this simple API lies considerable complexity.

The applicant's name is passed into the Mortgage constructor function. Then the `applyFor` method is called with the requested loan amount. Internally, this method uses services from 3 separate subsystems that are complex and possibly take some time to process; they are `Bank`, `Credit`, and `Background`.

Based on several criteria (bank statements, credit reports, and criminal background) the applicant is either accepted or denied for the requested loan.

```
var Mortgage = function(name) {
    this.name = name;
}

Mortgage.prototype = {
    applyFor: function(amount) {

        // access multiple subsystems...

        var result = "approved";
        if (!new Bank().verify(this.name, amount)) {
            result = "denied";
        } else if (!new Credit().get(this.name)) {
            result = "denied";
        } else if (!new Background().check(this.name)) {
            result = "denied";
        }

        return this.name + " has been " + result +
               " for a " + amount + " mortgage";
    }
}

var Bank = function() {
    this.verify = function(name, amount) {
        // complex logic ...
        return true;
    }
}
var Credit = function() {
    this.get = function(name) {
        // complex logic ...
        return true;
    }
}
var Background = function() {
    this.check = function(name) {
        // complex logic ...
        return true;
    }
}


function run() {

    var mortgage = new Mortgage("Joan Templeton");
    var result = mortgage.applyFor("$100,000");

    alert(result);
}
```

JavaScript Optimized Code

The Namespace pattern is applied to keep the code out of the global namespace. Our namespace is named `Patterns.Classic`. A Revealing Module named `Facade` returns (i.e. reveals) only a single item:

the `Mortgage` constructor function and by associations its prototype. All other sub-systems which include `Bank`, `Credit`, and `Background` are maintained in the module's closure and hidden from view and access.

The `Patterns` object contains the `namespace` function which constructs namespaces non-destructively, that is, if a name already exists it won't overwrite it.

The `log` function is a helper which collects and displays results.

```
var Patterns = {
    namespace: function (name) {
        var parts = name.split(".");
        var ns = this;

        for (var i = 0, len = parts.length; i < len; i++) {
            ns[parts[i]] = ns[parts[i]] || {};
            ns = ns[parts[i]];
        }

        return ns;
    }
};

Patterns.namespace("Classic").Facade = (function () {

    var Bank = function () {
        this.verify = function (name, amount) {
            // complex logic ...
            return true;
        }
    }
    var Credit = function () {
        this.get = function (name) {
            // complex logic ...
            return true;
        }
    }
    var Background = function () {
        this.check = function (name) {
            // complex logic ...
            return true;
        }
    }
    var Mortgage = function (name) {
        this.name = name;
    }

    Mortgage.prototype = {
        applyFor: function (amount) {

            // access multiple subsystems...
            var result = "approved";
            if (!new Bank().verify(this.name, amount)) {
                result = "denied";
            } else if (!new Credit().get(this.name)) {
                result = "denied";
            } else if (!new Background().check(this.name)) {
                result = "denied";
            }

            return this.name + " has been " + result +
                    " for a " + amount + " mortgage";
        }
    }

    return {
        Mortgage: Mortgage
    };

})();


function run() {

    var facade = Patterns.Classic.Facade;

    var mortgage = new facade.Mortgage("Joan Templeton");
    var result = mortgage.applyFor("$100,000");

    alert(result);
}
```

Flyweight

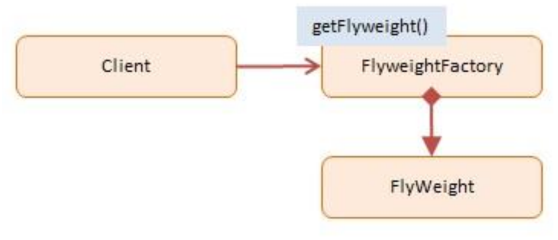
The Flyweight pattern conserves memory by sharing large numbers of fine-grained objects efficiently. Shared flyweight objects are immutable, that is, they cannot be changed as they represent the characteristics that are shared with other objects.

Essentially Flyweight is an 'object normalization technique' in which common properties are factored out into shared flyweight objects. (Note: the idea is similar to data model normalization, a process in which the modeler attempts to minimize redundancy).

An example of the Flyweight Pattern is within the JavaScript engine itself which maintains a list of immutable strings that are shared across the application.

Other examples include characters and line-styles in a word processor, or 'digit receivers' in a public switched telephone network application. You will find flyweights mostly in utility type applications such as word processors, graphics programs, and network apps; they are less often used in data-driven business type applications.

Diagram



Participants

The objects participating in this pattern are:

- Client -- In sample code: Computer
    - calls into FlyweightFactory to obtain flyweight objects
- FlyweightFactory -- In sample code: FlyweightFactory
    - creates and manages flyweight objects
    - if requested, and a flyweight does not exist, it will create one
    - stores newly created flyweights for future requests
- Flyweight -- In sample code: Flyweight
    - maintains intrinsic data to be shared across the application

JavaScript Code

In our example code we are building computers. Many models, makes, and processor combinations are common, so these characteristics are factored out and shared by Flyweight objects.

The `FlyweightFactory` maintains a pool of `Flyweight` objects. When a request arrives for a `Flyweight` object the `FlyweightFactory` will check if one already exists; if not, a new one will be created and stored for future reference. All subsequent requests for that same computer will return the stored `Flyweight` object.

Several different computers are added to a `ComputerCollection`. At the end we have a list of 7 `Computer` objects that share only 2 `Flyweights`. These are small savings, but with large datasets the memory savings can be significant.

The `log` function is a helper which collects and displays results.

```
function Flyweight (make, model, processor) {
    this.make = make;
    this.model = model;
    this.processor = processor;
};
```

```
var FlyWeightFactory = (function () {
    var flyweights = {};

    return {
        get: function (make, model, processor) {

            if (!flyweights[make + model]) {
                flyweights[make + model] =
                    new Flyweight(make, model, processor);
            }

            return flyweights[make + model];
        },
        getCount: function () {
            var count = 0;
            for (var f in flyweights) count++;
            return count;
        }
    }
})();

function ComputerCollection () {
    var computers = {};
    var count = 0;

    return {
        add: function (make, model, processor, memory, tag) {
            computers[tag] =
                new Computer(make, model, processor, memory, tag);
            count++;
        },
        get: function (tag) {
            return computers[tag];
        },
        getCount: function () {
            return count;
        }
    };
}

var Computer = function (make, model, processor, memory, tag) {
    this.flyweight = FlyWeightFactory.get(make, model, processor);
    this.memory = memory;
    this.tag = tag;

    this.getMake = function () {
        return this.flyweight.make;
    }

    // ...
}

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


function run() {

    var computers = new ComputerCollection();

    computers.add("Dell", "Studio XPS", "Intel", "5G", "Y755P");
    computers.add("Dell", "Studio XPS", "Intel", "6G", "X997T");
    computers.add("Dell", "Studio XPS", "Intel", "2G", "U8U80");
    computers.add("Dell", "Studio XPS", "Intel", "2G", "NT777");
    computers.add("Dell", "Studio XPS", "Intel", "2G", "0J88A");
    computers.add("HP", "Envy", "Intel", "4G", "CNU883701");
    computers.add("HP", "Envy", "Intel", "2G", "TXU003283");

    log.add("Computers: " + computers.getCount());
    log.add("Flyweights: " + FlyWeightFactory.getCount());

    log.show();
}
```

JavaScript Optimized Code

The Namespace pattern is applied to keep the code out of the global namespace. Our namespace is named `Patterns.Classic`. A Revealing Module named `Flyweight` returns (i.e. reveals) two items: `create`, which, in fact, is a Factory Method pattern, and `ComputerCollection` which is list of computers we are managing.

Two pre-fabricated prototype objects (flyweights) have been created, one for Dell and one for HP, each with their own id and other values. The method `create` determines which prototype object to assign to the new `Computer` object. We have implemented the classic Flyweight pattern using JavaScript's built-in prototypal inheritance system.

The `Patterns` object contains the `namespace` function which constructs namespaces non-destructively, that is, if a name already exists it won't overwrite it.

The `log` function is a helper which collects and displays results.

```
var Patterns = {
    namespace: function (name) {
        var parts = name.split(".");
        var ns = this;

        for (var i = 0, len = parts.length; i < len; i++) {
            ns[parts[i]] = ns[parts[i]] || {};
            ns = ns[parts[i]];
        }

        return ns;
    }
};

Patterns.namespace("Classic").Flyweight = (function () {

    // prototype flyweights
    var Proto = function (id, make, model, processor) {
        this.id = id;
        this.make = make;
        this.model = model;
        this.processor = processor;
    }
    var protoDell = new Proto(1, "Dell", "Studio XPS", "Intel");
    var protoHp = new Proto(2, "HP", "Envy", "Intel");

    var Computer = function (memory, tag) {
        this.memory = memory;
        this.tag = tag;
    };

    function create(make, model, processor, memory, tag) {

        if (make === "Dell" && model === "Studio XPS") {
            Computer.prototype = protoDell;
        } else if (make === "HP" && model === "Envy") {
            Computer.prototype = protoHp;
        }

        return new Computer(memory, tag);
    }

    var ComputerCollection = function () {
        var computers = {};
        var count = 0;

        return {
            add: function (computer) {
                computers[computer.tag] = computer;
                count++;
            },
            get: function (tag) {
                return computers[tag];
            },
            getCount: function () {
                return count;
            },
            getPrototypeCount: function () {
                var types = {};
                for (var tag in computers) types[computers[tag].id] = true;

                var count = 0;
                for (var t in types) count++;
                return count;
            }
        };
    }

    return {
        create: create,
        ComputerCollection: ComputerCollection
    };

})();

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


function run1() {

    var flyweight = Patterns.Classic.Flyweight;

    var computers = new flyweight.ComputerCollection();

    computers.add(flyweight.create("Dell", "Studio XPS", "Intel", "5G", "Y755P"));
    computers.add(flyweight.create("Dell", "Studio XPS", "Intel", "6G", "X997T"));
    computers.add(flyweight.create("Dell", "Studio XPS", "Intel", "2G", "U8U80"));
    computers.add(flyweight.create("Dell", "Studio XPS", "Intel", "2G", "NT777"));
    computers.add(flyweight.create("Dell", "Studio XPS", "Intel", "2G", "0J88A"));
    computers.add(flyweight.create("HP", "Envy", "Intel", "4G", "CNU883701"));
    computers.add(flyweight.create("HP", "Envy", "Intel", "2G", "TXU003283"));

    log.add("Computers: " + computers.getCount());
    log.add("Prototypes: " + computers.getPrototypeCount());

    log.show();
}
```

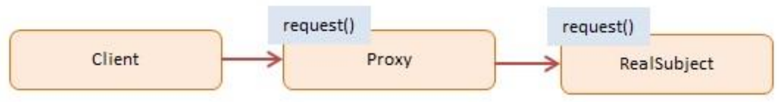
Proxy

The Proxy pattern provides a surrogate or placeholder object for another object and controls access to this other object.

In object-oriented programming, objects do the work they advertise through their interface (properties and methods). Clients of these objects expect this work to be done quickly and efficiently. However, there are situations where an object is severely constrained and cannot live up to its responsibility. Typically this occurs when there is a dependency on a remote resource (resulting in network latency) or when an object takes a long time to load.

In situations like these you apply the Proxy pattern and create a proxy object that 'stands in' for the original object. The Proxy forwards the request to a target object. The interface of the Proxy object is the same as the original object and clients may not even be aware they are dealing with a proxy rather than the real object.

Diagram



Participants

The objects participating in this pattern are:

- Client -- In sample code: the run() function
    - calls Proxy to request an operation
- Proxy -- In sample code: GeoProxy
    - provides an interface similar to the real object
    - maintains a reference that lets the proxy access the real object
    - handles requests and forwards these to the real object
- RealSubject -- In sample code: GeoCoder
    - defines the real object for which service is requested

JavaScript Code

The `GeoCoder` object simulates the Google Maps Geocoding service. In geocoding you provide a location (a place on the earth) and it will return its latitude/longitude (latlng). Our `GeoCoder` can resolve only 4 locations, but in reality there are millions, because it involves countries, cities, and streets.

The developer decides to implement a Proxy object because `GeoCoder` is relatively slow. The proxy object is called `GeoProxy`. It is known that in our hypothetical web app many repeated requests (for the same location) are coming in. To speed things up `GeoProxy` caches frequently requested locations. If a location is not already cached it goes out to the real `GeoCoder` service and stores the results in cache.

Several city locations are queried and many of these are for the same city. `GeoProxy` builds up its cache while supporting these calls. At the end `GeoProxy` has processed 11 requests but had to go out to GeoCoder only 3 times. Notice that the client program has no idea it is dealing with a proxy object because it calls the same Google Maps Geocoding interface with the standard getLatLng method.

The `log` function is a helper which collects and displays results.

```
function GeoCoder() {
    this.getLatLng = function(address) {

        if (address === "Amsterdam") {
            return "52.3700° N, 4.8900° E";
        } else if (address === "London") {
            return "51.5171° N, 0.1062° W";
        } else if (address === "Paris") {
            return "48.8742° N, 2.3470° E";
        } else if (address === "Berlin") {
            return "52.5233° N, 13.4127° E";
        } else {
            return "";
        }
    };
}

function GeoProxy() {
    var geocoder = new GeoCoder();
    var geocache = {};

    return {
        getLatLng: function(address) {
            if (!geocache[address]) {
                geocache[address] = geocoder.getLatLng(address);
            }

            log.add(address + ": " + geocache[address]);
            return geocache[address];
        },
        getCount: function() {
            var count = 0;
            for (var code in geocache) { count++; }
            return count;
        }
    };
};

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


function run() {

    var geo = new GeoProxy();

    // geolocation requests
    geo.getLatLng("Paris");
    geo.getLatLng("London");
    geo.getLatLng("London");
    geo.getLatLng("London");
    geo.getLatLng("London");
    geo.getLatLng("Amsterdam");
    geo.getLatLng("Amsterdam");
    geo.getLatLng("Amsterdam");
    geo.getLatLng("Amsterdam");
    geo.getLatLng("London");
    geo.getLatLng("London");

    log.add("\nCache size: " + geo.getCount());
    log.show();
}
```

JavaScript Optimized Code

The Namespace pattern is applied to keep the code out of the global namespace. Our namespace is named `Patterns.Classic`. A Revealing Module named `Proxy` returns (i.e. reveals) only a single item: the `Geo` constructor function which in turn supports two public methods: `getLatLng` and `getCount`. The public `GeoCoder` interface remains public because it simulates Google Maps geocoding functionality.

The `Patterns` object contains the `namespace` function which constructs namespaces non-destructively, that is, if a name already exists it won't overwrite it.

The `log` function is a helper which collects and displays results.

```
var Patterns = {
    namespace: function (name) {
        var parts = name.split(".");
        var ns = this;

        for (var i = 0, len = parts.length; i < len; i++) {
            ns[parts[i]] = ns[parts[i]] || {};
            ns = ns[parts[i]];
        }

        return ns;
    }
};

Patterns.namespace("Classic").Proxy = (function () {

    var Geo = function () {
        var geocoder = new GeoCoder();
        var geocache = {};

        return {
            getLatLng: function (address) {
                if (!geocache[address]) {
                    geocache[address] = geocoder.getLatLng(address);
                }

                log.add(address + ": " + geocache[address]);
                return geocache[address];
            },
            getCount: function () {
                var count = 0;
                for (var code in geocache) { count++; }
                return count;
            }
        };
    };

    return {
        Geo: Geo
    };
})();

// public interface
function GeoCoder() {
    this.getLatLng = function (address) {

        if (address === "Amsterdam") {
            return "52.3700° N, 4.8900° E";
        } else if (address === "London") {
            return "51.5171° N, 0.1062° W";
        } else if (address === "Paris") {
            return "48.8742° N, 2.3470° E";
        } else if (address === "Berlin") {
            return "52.5233° N, 13.4127° E";
        } else {
            return "";
        }
    };
}

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


function run() {

    var proxy = Patterns.Classic.Proxy;

    var geo = new proxy.Geo();

    // geolocation requests
    geo.getLatLng("Paris");
    geo.getLatLng("London");
    geo.getLatLng("London");
    geo.getLatLng("London");
    geo.getLatLng("London");
    geo.getLatLng("Amsterdam");
    geo.getLatLng("Amsterdam");
    geo.getLatLng("Amsterdam");
    geo.getLatLng("Amsterdam");
    geo.getLatLng("London");
    geo.getLatLng("London");

    log.add("\nCache size: " + geo.getCount());
    log.show();

}
```

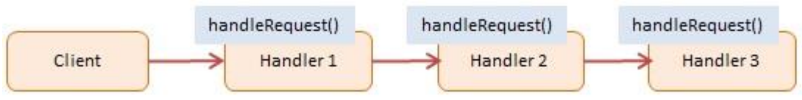
Chain of Responsibility

The Chain of Responsibility pattern provides a chain of loosely coupled objects one of which can satisfy a request. This pattern is essentially a linear search for an object that can handle a particular request.

An example of a chain-of-responsibility is event-bubbling in which an event propagates through a series of nested controls one of which may choose to handle the event.

The Chain of Responsibility patterns is related to the Chaining Pattern which is frequently used in JavaScript (jQuery makes extensive use of this pattern). To learn more about the Chaining patterns view the Modern Patterns section.

Diagram



Participants

The objects participating in this pattern are:

- Client -- In sample code: Request
    - initiates the request to a chain of handler objects
- Handler -- In sample code: Request.get() method
    - defines an interface for handling the requests
    - implements the successor link (returning 'this')

JavaScript Code

This example differs slightly from the classic Chain of Responsibility pattern in that not one, but all handlers participate in handling the request.

The code demonstrates an elegant solution to a money dispensing machine problem. Say, a customer requires $247 from an ATM machine. What is the combination of bank notes ($100, $50, $20, $10, $5, $1) that satisfies that request?

A `Request` is created with the amount requested. Next, a series of `get` calls are chained together, each one handling a particular denomination. Each handler determines the number of bank notes dispensed and subtracts this amount from the remaining amount. The request object is passed through the chain by returning `this` in the `get` method.

The `log` function is a helper which collects and displays results.

```
var Request = function(amount) {
    this.amount = amount;

    log.add("Requested: $" + amount + "\n");
}

Request.prototype = {
    get: function(bill) {
        var count = Math.floor(this.amount / bill);
        this.amount -= count * bill;

        log.write("Dispense " + count + " $" + bill + " bills");

        return this;
    }
}

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


function run() {

    var request = new Request(378);
    request.get(100).get(50).get(20).get(10).get(5).get(1);

    log.show();
}
```

JavaScript Optimized Code

The Namespace pattern is applied to keep the code out of the global namespace. Our namespace is named `Patterns.Classic`. A Revealing Module named `Chain` returns (i.e. reveals) only a single item: the `Request` constructor function and its prototype by association.

The `Patterns` object contains the `namespace` function which constructs namespaces non-destructively, that is, if a name already exists it won't overwrite it.

The `log` function is a helper which collects and displays results.

```
var Patterns = {
    namespace: function (name) {
        var parts = name.split(".");
        var ns = this;

        for (var i = 0, len = parts.length; i < len; i++) {
            ns[parts[i]] = ns[parts[i]] || {};
            ns = ns[parts[i]];
        }

        return ns;
    }
};

Patterns.namespace("Classic").Chain = (function () {
    var Request = function (amount) {
        this.amount = amount;

        log.add("Requested: $" + amount + "\n");
    }

    Request.prototype = {
        get: function (bill) {
            var count = Math.floor(this.amount / bill);
            this.amount -= count * bill;

            log.add("Dispense " + count + " $" + bill + " bills");

            return this;
        }
    }

    return {
        Request: Request
    }
})();

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

function run() {

    var chain = Patterns.Classic.Chain;

    var request = new chain.Request(378);
    request.get(100).get(50).get(20).get(10).get(5).get(1);

    log.show();
}
```

Command

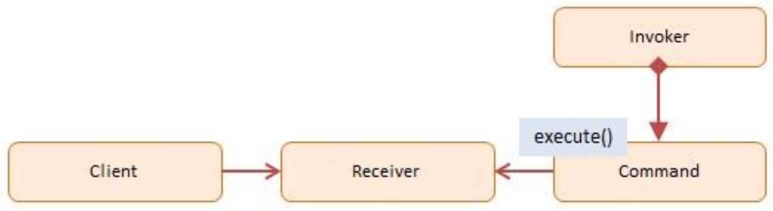
The Command pattern encapsulates actions as objects. Command objects allow for loosely coupled systems by separating the objects that issue a request from the objects that actually process the request. These requests are called events and the code that processes the requests are called event handlers.

Suppose you are building an application that supports the Cut, Copy, and Paste clipboard actions. These actions can be triggered in different ways throughout the app: by a menu system, a context menu (e.g. by right clicking on a textbox), or by a keyboard shortcut.

Command objects allow you to centralize the processing of these actions, one for each operation. So, you need only one Command for processing all Cut requests, one for all Copy requests, and one for all Paste requests.

Because commands centralize all processing, they are also frequently involved in handling Undo functionality for the entire application.

Diagram



Participants

The objects participating in this pattern are:

- Client -- In sample code: the run() function

  - references the Receiver objects
- Receiver -- In sample code: Calculator
  - knows how to carry out the operation associated with the command
  - (optionally) maintains a history of executed commands
- Command -- In sample code: Command
  - maintains information about the action to be taken
- Invoker -- In our sample code: the user pushing the buttons
  - asks to carry out the request

JavaScript Code

In our example we have a calculator with 4 basic operations: `add`, `subtract`, `multiply` and `divide`. Each operation is encapsulated by a `Command` object.

The `Calculator` maintains a stack of commands. Each new command is executed and pushed onto the stack. When an undo request arrives, it simply pops the last command from the stack and executes the reverse action.

JavaScript's function objects (and callbacks) are native command objects. They can be passed around like objects; in fact, they are true objects. To learn more about JavaScript's eventing system and how callbacks work please review the Modern Patterns section.

The `log` function is a helper which collects and displays results.

```
function add(x, y) { return x + y; }
function sub(x, y) { return x - y; }
function mul(x, y) { return x * y; }
function div(x, y) { return x / y; }

var Command = function (execute, undo, value) {
    this.execute = execute;
    this.undo = undo;
    this.value = value;
}

var AddCommand = function (value) {
    return new Command(add, sub, value);
};

var SubCommand = function (value) {
    return new Command(sub, add, value);
};

var MulCommand = function (value) {
    return new Command(mul, div, value);
};

var DivCommand = function (value) {
    return new Command(div, mul, value);
```

```
};

var Calculator = function () {
    var current = 0;
    var commands = [];

    function action(command) {
        var name = command.execute.toString().substr(9, 3);
        return name.charAt(0).toUpperCase() + name.slice(1);
    }

    return {
        execute: function (command) {

            current = command.execute(current, command.value);
            commands.push(command);

            log.add(action(command) + ": " + command.value);
        },
        undo: function () {
            var command = commands.pop();
            current = command.undo(current, command.value);

            log.add("Undo " + action(command) + ": " + command.value);
        },
        getCurrentValue: function () {
            return current;
        }
    }
}

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


function run() {

    var calculator = new Calculator();

    // issue commands
    calculator.execute(new AddCommand(100));
    calculator.execute(new SubCommand(24));
    calculator.execute(new MulCommand(6));
    calculator.execute(new DivCommand(2));

    // reverse last two commands
    calculator.undo();
    calculator.undo();
    log.add("\nValue: " + calculator.getCurrentValue());

    log.show();
}
```

JavaScript Optimized Code

The Namespace pattern is applied to keep the code out of the global namespace. Our namespace is named `Patterns.Classic`. A Revealing Module named `Command` returns (i.e. reveals) five public items: `AddCommand`, `SubCommand`, `MulCommand`, `DivCommand` and the `Calculator` constructor function. All other items are kept private in the module.

The `Patterns` object contains the `namespace` function which constructs namespaces non-destructively, that is, if a name already exists it won't overwrite it.

The `log` function is a helper which collects and displays results.

```
var Patterns = {
    namespace: function (name) {
        var parts = name.split(".");
        var ns = this;

        for (var i = 0, len = parts.length; i < len; i++) {
            ns[parts[i]] = ns[parts[i]] || {};
            ns = ns[parts[i]];
        }

        return ns;
    }
};

Patterns.namespace("Classic").Command = (function () {

    function add(x, y) { return x + y; }
    function sub(x, y) { return x - y; }
    function mul(x, y) { return x * y; }
    function div(x, y) { return x / y; }

    var Command = function (execute, undo, value) {
        this.execute = execute;
        this.undo = undo;
        this.value = value;
    }

    return {
        AddCommand: function (value) {
            return new Command(add, sub, value);
        },

        SubCommand: function (value) {
            return new Command(sub, add, value);
        },

        MulCommand: function (value) {
            return new Command(mul, div, value);
        },

        DivCommand: function (value) {
            return new Command(div, mul, value);
        },

        Calculator: function () {
            var current = 0;
            var commands = [];

            function action(command) {
                var name = command.execute.toString().substr(9, 3);
                return name.charAt(0).toUpperCase() + name.slice(1);
            }

            return {
                execute: function (command) {

                    current = command.execute(current, command.value);
                    commands.push(command);

                    log.add(action(command) + ": " + command.value);
                },
                undo: function () {
                    var command = commands.pop();
                    current = command.undo(current, command.value);

                    log.add("Undo " + action(command) + ": " + command.value);
                },
                getCurrentValue: function () {
                    return current;
                }
            }
        }
    };

})();

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


function run() {

    var command = Patterns.Classic.Command;
    var calculator = new command.Calculator();

    // issue commands
    calculator.execute(new command.AddCommand(100));
    calculator.execute(new command.SubCommand(24));
    calculator.execute(new command.MulCommand(6));
    calculator.execute(new command.DivCommand(2));

    // reverse last two commands
    calculator.undo();
    calculator.undo();

    log.add("\nValue: " + calculator.getCurrentValue());

    log.show();
}
```

Interpreter

The Interpreter pattern offers a scripting language that allows end-users to customize their solution.

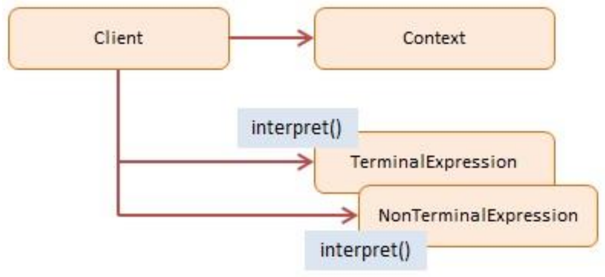
Some applications are so complex that they require advanced configuration. You could offer a basic scripting language which allows the end-user to manipulate the application through simple instructions. The Interpreter pattern allows you to solve this particular problem – that of creating a simple scripting language.

Certain types of problems lend themselves to be characterized by a language. This language describes the problem domain which should be well-understood and well-defined. To implement this you need to map the language to a grammar. Grammars are usually hierarchical tree-like structures that step through multiple levels and then end up with terminal nodes (also called literals).

Problems like this, expressed as a grammar, can be implemented using the Interpreter design pattern.

Today, if you really need this type of control in JavaScript it is probably easier to use a code generator like ANTLR which will allow you to build your own command interpreters based on a grammar that you provide.

Diagram



Participants

The objects participating in this pattern are:

- Client -- In sample code: the run() program.
    - builds (or is given) a syntax tree representing the grammar
    - establishes the initial context
    - invokes the interpret operations
- Context -- In sample code: Context
    - contains state information to the interpreter
- TerminalExpression -- In sample code: Expression
    - implements an interpret operation associated with terminal symbols in the grammar
    - one instance for each terminal expression in the sentence
- NonTerminalExpression -- In sample code: not used
    - implements an interpret operation associated for non-terminal symbols in the grammar

JavaScript Code

The objective of this example is to build an interpreter which translates roman numerals to decimal numbers: for example, XXXVI = 36.

The `Context` object maintains the input (the roman numeral) and the resulting output as it is being parsed and interpreted. The `Expression` object represents the nodes in the grammar tree; it supports the `interpret` method.

When running the program, a simple grammar tree is being built which then processes a roman numeral and translates it into a numeric.

```
var Context = function (input) {
    this.input = input;
    this.output = 0;
}

Context.prototype = {
    startsWith : function (str) {
        return this.input.substr(0, str.length) === str;
    }
}

var Expression = function (name, one, four, five, nine, multiplier) {
    this.name = name;
    this.one = one;
    this.four = four;
    this.five = five;
    this.nine = nine;
    this.multiplier = multiplier;
}

Expression.prototype = {
    interpret: function (context) {
        if (context.input.length == 0) {
            return;
        }
        else if (context.startsWith(this.nine)) {
            context.output += (9 * this.multiplier);
            context.input = context.input.substr(2);
        }
        else if (context.startsWith(this.four)) {
            context.output += (4 * this.multiplier);
            context.input = context.input.substr(2);
        }
        else if (context.startsWith(this.five)) {
            context.output += (5 * this.multiplier);
            context.input = context.input.substr(1);
        }

        while (context.startsWith(this.one)) {
            context.output += (1 * this.multiplier);
            context.input = context.input.substr(1);
        }
    }
}


function run() {

    var roman = "MCMXXVIII"
    var context = new Context(roman);
    var tree = [];

    tree.push(new Expression("thousand", "M", " " , " ", " " , 1000));
    tree.push(new Expression("hundred",  "C", "CD", "D", "CM", 100));
    tree.push(new Expression("ten",      "X", "XL", "L", "XC", 10));
    tree.push(new Expression("one",      "I", "IV", "V", "IX", 1));

    for (var i = 0, len = tree.length; i < len; i++) {
        tree[i].interpret(context);
    }

    alert(roman + " = " + context.output);
}
```

JavaScript Optimized Code

The Namespace pattern is applied to keep the code out of the global namespace. Our namespace is named `Patterns.Classic`. A Revealing Module named `Interpreter` returns (i.e. reveals) only a single item: the `evaluate` method. All other items are kept private in the module; OO encapsulation at its best.

The `Patterns` object contains the `namespace` function which constructs namespaces non-destructively, that is, if a name already exists it won't overwrite it.

The `log` function is a helper which collects and displays results.

```
var Patterns = {
    namespace: function (name) {
        var parts = name.split(".");
        var ns = this;

        for (var i = 0, len = parts.length; i < len; i++) {
            ns[parts[i]] = ns[parts[i]] || {};
            ns = ns[parts[i]];
        }

        return ns;
    }
};

Patterns.namespace("Classic").Interpreter = (function () {

    var Context = function (input) {
        this.input = input;
        this.output = 0;
    }

    Context.prototype = {
        startsWith: function (str) {
            return this.input.substr(0, str.length) === str;
        }
    }

    var Expression = function (name, one, four, five, nine, multiplier) {
        this.name = name;
        this.one = one;
        this.four = four;
        this.five = five;
        this.nine = nine;
        this.multiplier = multiplier;
    }

    Expression.prototype = {
        interpret: function (context) {
            if (context.input.length == 0) {
                return;
            }
            else if (context.startsWith(this.nine)) {
                context.output += (9 * this.multiplier);
                context.input = context.input.substr(2);
            }
            else if (context.startsWith(this.four)) {
                context.output += (4 * this.multiplier);
                context.input = context.input.substr(2);
            }
            else if (context.startsWith(this.five)) {
                context.output += (5 * this.multiplier);
                context.input = context.input.substr(1);
            }

            while (context.startsWith(this.one)) {
                context.output += (1 * this.multiplier);
                context.input = context.input.substr(1);
            }
        }
    }

    function evaluate(roman) {

        var tree = [];
        var context = new Context(roman);

        tree.push(new Expression("thousand", "M", " ", " ", " ", 1000));
        tree.push(new Expression("hundred", "C", "CD", "D", "CM", 100));
        tree.push(new Expression("ten", "X", "XL", "L", "XC", 10));
        tree.push(new Expression("one", "I", "IV", "V", "IX", 1));

        for (var i = 0, len = tree.length; i < len; i++) {
            tree[i].interpret(context);
        }

        return context.output;
    }

    return {
        evaluate: evaluate
    };

})();


function run() {

    var roman = "MCMXXVIII"
    var result = Patterns.Classic.Interpreter.evaluate(roman);

    alert(roman + " = " + result);
}
```

Iterator

The Iterator pattern allows clients to effectively loop over a collection of objects.

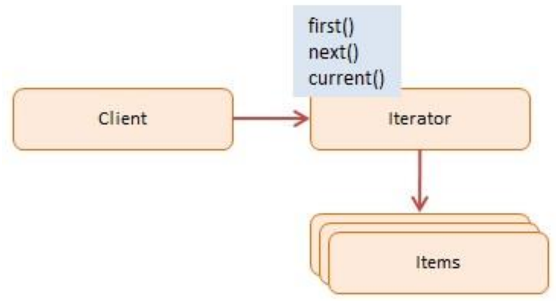
A common programming task is to traverse and manipulate a collection of objects. These collections may be stored as an array or perhaps something more complex, such as a tree or graph structure. In addition, you may need to access the items in the collection in a certain order, such as, front to back, back to front, depth first (as in tree searches), skip evenly numbered objects, etc.

The Iterator design pattern solves this problem by separating the collection of objects from the traversal of these objects by implementing a specialized iterator.

Today, many languages have Iterators built-in by supporting 'for-each'-type constructs and IEnumerable and IEnumerator interfaces. However, JavaScript only supports basic looping in the form of `for`, `for-in`, `while`, and `do while` statements.

The Iterator pattern allows JavaScript developers to design looping constructs that are far more flexible and sophisticated.

Diagram



Participants

The objects participating in this pattern are:

- Client -- In sample code: the run() function
    - references and invokes Iterator with collection of objects
- Iterator -- In sample code: Iterator
    - implements iterator interface with methods first(), next(), etc
    - keeps track of current position when traversing collection
- Items -- In sample code: Items
    - individual objects of the collection being traversed

JavaScript Code

The `Iterator` object maintains a reference to the collection and the current position. It also implements the 'standard' Iterator interface with methods like: `first`, `next`, `hasNext`, `reset`, and `each`.

Two looping methods are used: a built-in `for` loop and a newly created `each` method. The `for` loop uses the `first`, `hasNext`, and `next` methods to control the iteration. The `each` method does internally exactly the same (i.e. runs a for loop), but to the client the syntax has been greatly simplified.

The `log` function is a helper which collects and displays results.

```
var Iterator = function(items) {
    this.index = 0;
    this.items = items;
}

Iterator.prototype = {
    first: function() {
        this.reset();
        return this.next();
    },
    next: function() {
        return this.items[this.index++];
    },
    hasNext: function() {
        return this.index <= this.items.length;
    },
    reset: function() {
        this.index = 0;
    },
    each: function(callback) {
        for (var item = this.first(); this.hasNext(); item = this.next()) {
            callback(item);
        }
    }
}

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


function run() {

    var items = ["one", 2, "circle", true, "Applepie"];
    var iter = new Iterator(items);

    // using for loop
    for (var item = iter.first(); iter.hasNext(); item = iter.next()) {
        log.add(item);
    }

    log.add("");

    // using Iterator's each method

    iter.each(function(item) {
        log.add(item);
    });

    log.show();
}
```

JavaScript Optimized Code

The Namespace pattern is applied to keep the code out of the global namespace. Our namespace is named `Patterns.Classic`. A Revealing Module named `Iterator` returns (i.e. reveals) only a single item: the `Iterator` and by association its prototype object.

The `Patterns` object contains the `namespace` function which constructs namespaces non-destructively, that is, if a name already exists it won't overwrite it.

The `log` function is a helper which collects and displays results.

```
var Patterns = {
    namespace: function (name) {
        var parts = name.split(".");
        var ns = this;

        for (var i = 0, len = parts.length; i < len; i++) {
            ns[parts[i]] = ns[parts[i]] || {};
            ns = ns[parts[i]];
        }

        return ns;
    }
};

Patterns.namespace("Classic").Iterator = (function () {

    var Iterator = function (items) {
        this.index = 0;
        this.items = items;
    }

    Iterator.prototype = {
        first: function () {
            this.reset();
            return this.next();
        },
        next: function () {
            return this.items[this.index++];
        },
        hasNext: function () {
            return this.index <= this.items.length;
        },
        reset: function () {
            this.index = 0;
        },
        each: function (callback) {
            for (var item = this.first() ; this.hasNext() ; item = this.next()) {
                callback(item);
            }
        }
    }

    return {
        Iterator: Iterator
    }

})();

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


function run() {

    var items = ["one", 2, "circle", true, "Applepie"];
    var iter = new Patterns.Classic.Iterator.Iterator(items);

    // using for loop

    for (var item = iter.first() ; iter.hasNext() ; item = iter.next()) {
        log.add(item);
    }

    log.add("");

    // using Iterator's each method

    iter.each(function (item) {
        log.add(item);
    });

    log.show();
}
```

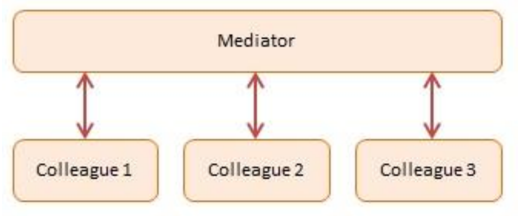
Mediator

The Mediator pattern provides central authority over a group of objects by encapsulating how these objects interact. This model is useful for scenarios where there is a need to manage complex conditions in which every object is aware of any state change in any other object in the group.

The Mediator patterns are useful in the development of complex forms. Take for example a page in which you enter options to make a flight reservation. A simple Mediator rule would be: you must enter a valid departure date, a valid return date, the return date must be after the departure date, a valid departure airport, a valid arrival airport, a valid number of travelers, and only then the Search button can be activated.

Another example of Mediator is that of a control tower on an airport coordinating arrivals and departures of airplanes.

Diagram



Participants

The objects participating in this pattern are:

- Mediator -- In sample code: Chatroom
  - defines an interface for communicating with Colleague objects
  - maintains references to Colleague objects
  - manages central control over operations
- Colleagues -- In sample code: Participants
  - objects that are being mediated by the Mediator
  - each instance maintains a reference to the Mediator

JavaScript Code

Four participants are joining in a chat session by registering with a `Chatroom` (the Mediator). Each participant is represented by a Participant object. Participants send messages to each other and the Chatroom handles the routing.

This example is simple, but other complex rules could have been added, such as a 'junk filter' to protect participants from receiving junk messages.

The `log` function is a helper which collects and displays results.

```
var Participant = function(name) {
    this.name = name;
    this.chatroom = null;
};

Participant.prototype = {
    send: function(message, to) {
        this.chatroom.send(message, this, to);
    },
    receive: function(message, from) {
        log.add(from.name + " to " + this.name + ": " + message);
    }
};

var Chatroom = function() {
    var participants = {};
    return {
        register: function(participant) {
            participants[participant.name] = participant;
            participant.chatroom = this;
        },
        send: function(message, from, to) {
            if (to) {                      // single message
                to.receive(message, from);
            } else {                       // broadcast message
                for (key in participants) {
                    if (participants[key] !== from) {
                        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.");
    john.send("Hey, no need to broadcast", yoko);
    paul.send("Ha, I heard that!");
    ringo.send("Paul, what do you think?", paul);

    log.show();
}
```

JavaScript Optimized Code

The Namespace pattern is applied to keep the code out of the global namespace. Our namespace is named `Patterns.Classic`. A Revealing Module named `Mediator` returns (i.e. reveals) a single item: `Chatroom`. The Chatroom's `register` method now creates and returns the newly created Participant instance. This allows us to keep to code in the run method simple and compact.

The `Patterns` object contains the `namespace` function which constructs namespaces non-destructively, that is, if a name already exists it won't overwrite it.

The `log` function is a helper which collects and displays results.

```
var Patterns = {
    namespace: function (name) {
        var parts = name.split(".");
        var ns = this;

        for (var i = 0, len = parts.length; i < len; i++) {
            ns[parts[i]] = ns[parts[i]] || {};
            ns = ns[parts[i]];
        }

        return ns;
    }
};

Patterns.namespace("Classic").Mediator = (function () {
    var Participant = function (name) {
        this.name = name;
        this.chatroom = null;
    };
    Participant.prototype = {
        send: function (message, to) {
            this.chatroom.send(message, this, to);
        },
        receive: function (message, from) {
            log.add(from.name + " to " + this.name + ": " + message);
        }
    };

    var Chatroom = function () {
        var participants = {};
        return {
            register: function (name) {
                var participant = new Participant(name);
                participants[participant.name] = participant;
                participant.chatroom = this;

                return participant;
            },
            send: function (message, from, to) {
                if (to) {                       // single message
                    to.receive(message, from);
                } else {                        // broadcast message
                    for (key in participants) {
                        if (participants[key] !== from) {
                            participants[key].receive(message, from);
                        }
                    }
                }
            }
        };
    };

    return {
        Chatroom: Chatroom
    }
})();


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


function run() {

    var chatroom = new Patterns.Classic.Mediator.Chatroom();

    var yoko = chatroom.register("Yoko");
    var john = chatroom.register("John");
    var paul = chatroom.register("Paul");
    var ringo = chatroom.register("Ringo");

    yoko.send("All you need is love.");
    yoko.send("I love you John.");
    john.send("Hey, no need to broadcast", yoko);
    paul.send("Ha, I heard that!");
    ringo.send("Paul, what do you think?", paul);

    log.show();
}
```

Memento

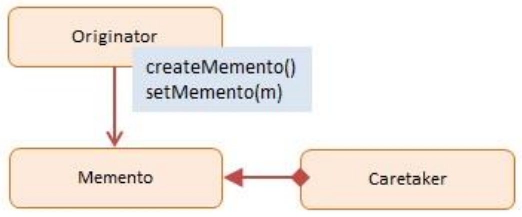
The Memento pattern provides temporary storage as well as restoration of an object. The mechanism in which you store the object’s state depends on the required duration of persistence, which may vary.

You could view a database as an implementation of the Memento design pattern in which objects are persisted and restored. However, the most common reason for using this pattern is to capture a snapshot of an object’s state so that any subsequent changes can be undone easily if necessary.

Essentially, a Memento is a small repository that stores an object’s state. Scenarios in which you may want to restore an object into a state that existed previously include: saving and restoring the state of a player in a computer game or the implementation of an undo operation in a database.

In JavaScript Mementos are easily implemented by serializing and de-serializing objects with JSON.

Diagram



Participants

The objects participating in this pattern are:

- Originator -- In sample code: Person
    - implements interface to create and restore mementos of itself
        - -- in sample code: hydrate and dehydrate
    - the object which state is temporary being saved and restored
- Memento -- In sample code: JSON representation of Person

    - internal state of the Originator object in some storage format
- CareTaker -- In sample code: CareTaker
    - responsible for storing mementos
    - just a repository; does not make changes to mementos

JavaScript Code

The sample code creates two persons named Mike and John using the `Person` constructor function. Next, their mementos are created which are maintained by the `CareTaker` object.

We assign Mike and John bogus names before restoring them from their mementos. Following the restoration we confirm that the person objects are back to their original state with valid names.

The Memento pattern itself with CareTaker etc. is rarely used in JavaScript. However, JSON is a highly effective data format that is extremely useful in many different data exchange scenarios.

```
var Person = function (name, street, city, state) {
    this.name = name;
    this.street = street;
    this.city = city;
    this.state = state;
}

Person.prototype = {
    hydrate: function () {

        var memento = JSON.stringify(this);
        return memento;
    },
    dehydrate: function (memento) {

        var m = JSON.parse(memento);
        for (var prop in m) this[prop] = m[prop];
    }
}

var CareTaker = function () {

    this.mementos = {};

    this.add = function (key, memento) {
        this.mementos[key] = memento;
    },
    this.get = function (key) {
        return this.mementos[key];
    }
}

function run() {

    var mike = new Person("Mike Foley", "1112 Main", "Dallas", "TX");
    var john = new Person("John Wang", "48th Street", "San Jose", "CA");

    var caretaker = new CareTaker();

    // save state

    caretaker.add(1, mike.hydrate());
    caretaker.add(2, john.hydrate());

    // mess up their names

    mike.name = "King Kong";
    john.name = "Superman";

    // restore original state

    mike.dehydrate(caretaker.get(1));
    john.dehydrate(caretaker.get(2));

    log.add(mike.name);
    log.add(john.name);

    log.show();
}
```

JavaScript Optimized Code

The Namespace pattern is applied to keep the code out of the global namespace. Our namespace is name `Patterns.Classic`. A Revealing Module named `Memento` returns (i.e. reveals) only a single item: the `makeHydratable` method.

In this implementation there is no need for a Caretaker; we simply added the hydrated Memento dynamically to an object. Then in the dehydrate method, the values are simply written back to the instance itself.

The `Patterns` object contains the `namespace` function which constructs namespaces non-destructively, that is, if a name already exists it won't overwrite it.

The `log` function is a helper which collects and displays results.

```
var Patterns = {
    namespace: function (name) {
        var parts = name.split(".");
        var ns = this;

        for (var i = 0, len = parts.length; i < len; i++) {
            ns[parts[i]] = ns[parts[i]] || {};
            ns = ns[parts[i]];
        }

        return ns;
    }
};

Patterns.namespace("Classic").Memento = (function () {

    var hydratable = {
        hydrate: function () {
            this.memento = JSON.stringify(this);
        },
        dehydrate: function () {

            if (this.memento) {
                var m = JSON.parse(this.memento);
                for (var prop in m) this[prop] = m[prop];

                this.memento = null;
            }
        },

        memento: null
    }

    var makeHydratable = function (obj) {
        for (var prop in hydratable) {
            obj[prop] = hydratable[prop];
        }
        return obj;
    };

    return {
        makeHydratable: makeHydratable
    }


})();

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


function run() {

    var Person = function (name, street, city, state) {
        this.name = name;
        this.street = street;
        this.city = city;
        this.state = state;
    }
    var mike = new Person("Mike Foley", "1112 Main", "Dallas", "TX");
    var john = new Person("John Wang", "48th Street", "San Jose", "CA");

    Patterns.Classic.Memento.makeHydratable(mike);
    Patterns.Classic.Memento.makeHydratable(john);

    // save state

    mike.hydrate();
    john.hydrate();

    // mess up their names

    mike.name = "King Kong";
    john.name = "Superman";

    // restore original state

    mike.dehydrate();
    john.dehydrate();

    log.add(mike.name);
    log.add(john.name);

    log.show();
}
```

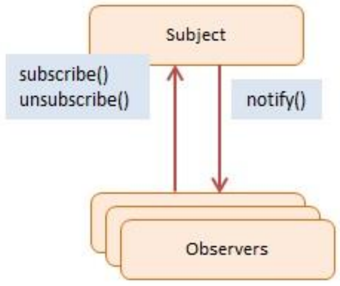
Observer

The Observer pattern offers a subscription model in which objects subscribe to an event and get notified when the event occurs. This pattern is the cornerstone of event driven programming, including JavaScript. The Observer pattern facilitates good object-oriented design and promotes loose coupling.

When building web apps you end up writing many event handlers. Event handlers are functions that will be notified when a certain event fires. These notifications optionally receive an event argument with details about the event (for example the x and y position of the mouse at a click event).

The event and event-handler paradigm in JavaScript is the manifestation of the Observer design pattern. Another name for the Observer pattern is Pub/Sub, short for Publication/Subscription.

Diagram



Participants

The objects participating in this pattern are:

- Subject -- In sample code: Click
    - maintains a list of observers. Any number of Observer objects may observe a Subject
    - implements an interface that lets observer objects subscribe or unsubscribe
    - sends a notification to its observers when its state changes
- Observers -- In sample code: clickHandler
    - has a function signature that can be invoked when Subject changes (i.e. event occurs)

JavaScript Code

The `Click` object represents the Subject. The `clickHandler` function is the subscribing Observer. This handler subscribes, unsubscribes, and then subscribes itself while events are firing. It gets notified only of events #1 and #3.

Notice that the `fire` method accepts two arguments. The first one has details about the event and the second one is the context, that is, the `this` value for when the event handlers are called. If no context is provided then `this` will be bound to the global object (window).

The `log` function is a helper which collects and displays results.

```
function Click() {
    this.handlers = [];  // observers
}

Click.prototype = {
    subscribe: function(fn) {
        this.handlers.push(fn);
    },

    unsubscribe: function(fn) {
        this.handlers = this.handlers.filter(
            function(item) {
                if (item !== fn) {
                    return item;
                }
            }
        );
    },

    fire: function(o, thisObj) {
        var scope = thisObj || window;
        this.handlers.forEach(function(item) {
            item.call(scope, o);
        });
    }
}

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


function run() {

    var clickHandler = function(item) {
        log.add("fired: " + item);
    };

    var click = new Click();

    click.subscribe(clickHandler);
    click.fire('event #1');

    click.unsubscribe(clickHandler);
    click.fire('event #2');

    click.subscribe(clickHandler);
    click.fire('event #3');

    log.show();
}
```

JavaScript Optimized Code

The JavaScript environment is event driven and the Observer pattern is built-in. There is no need to reinvent the wheel so we will use what is available. Unfortunately, the native event handling is different across browsers. But lucky for us, jQuery does a fantastic job of shielding developers from browser differences and it offers sophisticated event management and event handling, including custom events.

Here we have a div element on the page to which we attach three different event handlers using jQuery's `on` method. You will see that all three handlers execute when the 'poke' event gets triggered ('poke' is a custom event). At the end of the code we detach all event handlers with a single call to jQuery's `off` method.

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


function run() {

    var div = jQuery("#div");

    // attach three event handlers
    div.on("poke", function () { log.add("poke handler 1"); });
    div.on("poke", function () { log.add("poke handler 2"); });
    div.on("poke", function () { log.add("poke handler 3"); });

    // trigger event
    div.trigger('poke');

    log.show();

    // detach event handlers
    div.off("poke");
}
```

State

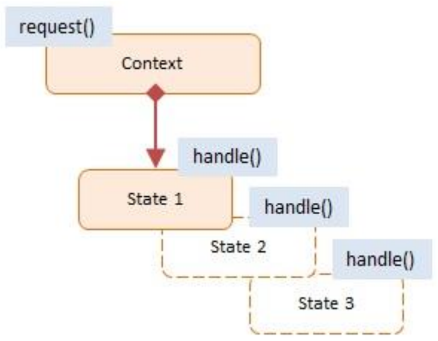
The State pattern provides state-specific logic to a limited set of objects in which each object represents a particular state. This is best explained with an example.

Say a customer places an online order for a TV. While this order is being processed it can in one of many states: New, Approved, Packed, Pending, Hold, Shipping, Completed, or Canceled. If all goes well the sequence will be New, Approved, Packed, Shipped, and Completed. However, at any point something unpredictable may happen, such as no inventory, breakage, or customer cancelation. If that happens the order needs to be handled appropriately.

Applying the State pattern to this scenario will provide you with 8 State objects, each with its own set of properties (state) and methods (i.e. the rules of acceptable state transitions). State machines are often implemented using the State pattern. These state machines simply have their State objects swapped out with another one when a state transition takes place.

Two other examples where the State pattern is useful include: vending machines that dispense products when a correct combination of coins is entered, and elevator logic which moves riders up or down depending on certain complex rules that attempt to minimize wait and ride times.

Diagram



Participants

The objects participating in this pattern are:

- Context -- In sample code: TrafficLight
    - exposes an interface that supports clients of the service
    - maintains a reference to a state object that defines the current state
    - allows State objects to change its current state to a different state
- State -- In sample code: Red, Yellow, Green
    - encapsulates the state values and associated behavior of the state

JavaScript Code

Our example is a traffic light (i.e. `TrafficLight` object) with 3 different states: `Red`, `Yellow` and `Green`, each with its own set of rules. The rules go like this: Say the traffic light is Red. After a delay the Red state changes to the Green state. Then, after another delay, the Green state changes to the Yellow state. After a very brief delay the Yellow state is changed to Red. And on and on.

It is important to note that it is the State object that determines the transition to the next state. And it is also the State object that changes the current state in the TrafficLight -- not the TrafficLight itself.

For demonstration purposes, a built-in counter limits the number of state changes, or else the program would run indefinitely.

The `log` function is a helper which collects and displays results.

```
var TrafficLight = function () {

    var count = 0;
    var currentState = new Red(this);
    this.change = function (state) {
        // limits number of changes
        if (count++ >= 10) return;

        currentState = state;
        currentState.go();
    };

    this.start = function () {
        currentState.go();
    };
}

var Red = function (light) {
    this.light = light;

    this.go = function () {
        log.add("Red --> for 1 minute");
        light.change(new Green(light));
    }
};

var Yellow = function (light) {
    this.light = light;

    this.go = function () {
        log.add("Yellow --> for 10 seconds");
        light.change(new Red(light));
    }
};

var Green = function (light) {
    this.light = light;

    this.go = function () {
        log.add("Green --> for 1 minute");
        light.change(new Yellow(light));
    }
};

// log helper

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


function run() {

    var light = new TrafficLight();
    light.start();

    log.show();
}
```

JavaScript Optimized Code

The Namespace pattern is applied to keep the code out of the global namespace. Our namespace is named `Patterns.Classic`. A Revealing Module named `State` returns (i.e. reveals) only a single item: the `TrafficLight` constructor function.

All three state items, that is, `Red`, `Yellow`, and `Green` are kept private in TrafficLight's closure. They have full access to the enclosing function which includes `change` through which they change the state. So the state objects do not need an explicit reference to TrafficLight anymore, they have implicit access. The module only exposes TrafficLight, which, in turn, only exposes the `start` method. This is a nice example of OO encapsulation and data hiding using JavaScript patterns.

The `Patterns` object contains the `namespace` function which constructs namespaces non-destructively, that is, if a name already exists it won't overwrite it.

The `log` function is a helper which collects and displays results.

```
var Patterns = {
    namespace: function (name) {
        var parts = name.split(".");
        var ns = this;

        for (var i = 0, len = parts.length; i < len; i++) {
            ns[parts[i]] = ns[parts[i]] || {};
            ns = ns[parts[i]];
        }

        return ns;
    }
};

Patterns.namespace("Classic").State = (function () {
    var TrafficLight = function () {

        var Red = function () {
            this.go = function () {
                log.add("Red --> for 1 minute");
                change(new Green());
            }
        };

        var Yellow = function () {
            this.go = function () {
                log.add("Yellow --> for 10 seconds");
                change(new Red());
            }
        };

        var Green = function () {
            this.go = function () {
                log.add("Green --> for 1 minute");
                change(new Yellow());
            }
        };

        var count = 0;
        var currentState = new Red(this);

        var change = function (state) {
            // limits number of changes
            if (count++ >= 10) return;

            currentState = state;
            currentState.go();
        };

        this.start = function () {
            currentState.go();
        };
    }

    return {
        TrafficLight: TrafficLight
    };

})();

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


function run() {

    var light = new Patterns.Classic.State.TrafficLight();
    light.start();

    log.show();
}
```

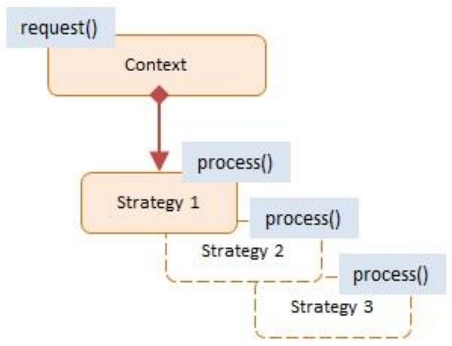
Strategy

The Strategy pattern encapsulates alternative algorithms (or strategies) for a particular task. It allows a method to be swapped out at runtime by any other method (strategy) without the client realizing it. Essentially, Strategy is a group of algorithms that are interchangeable.

Say we like to test the performance of different sorting algorithms to an array of numbers: shell sort, heap sort, bubble sort, quicksort, etc. Applying the Strategy pattern to these algorithms allows the test program to loop through all algorithms, simply by changing them at runtime and test each of these against the array. For Strategy to work all method signatures must be the same so that they can vary without the client program knowing about it.

In JavaScript the Strategy pattern is widely used as a plug-in mechanism when building extensible frameworks. This can be an extremely effective approach.

Diagram



Participants

The objects participating in this pattern are:

- Context -- In sample code: Shipping
  - maintains a reference to the current Strategy object
  - supports interface to allow clients to request Strategy calculations
  - allows clients to change Strategy
- Strategy -- In sample code: UPS, USPS, Fedex
  - implements the algorithm using the Strategy interface

JavaScript Code

In this example we have a product order that needs to be shipped from a warehouse to a customer. Different shipping companies are evaluated to determine the best price. This can be useful with shopping carts where customers select their shipping preferences and the selected Strategy returns the estimated cost.

`Shipping` is the Context and the 3 shipping companies `UPS`, `USPS`, and `Fedex` are the Strategies. The shipping companies (strategies) are changed 3 times and each time we calculate the cost of shipping. In a real-world scenario the calculate methods may call into the shipper's Web service. At the end we display the different costs.

The `log` function is a helper which collects and displays results.

```
var Shipping = function() {
    this.company = "";
};

Shipping.prototype = {

    setStrategy: function(company) {
        this.company = company;
    },
    calculate: function(package) {
        return this.company.calculate(package);
    }
};

var UPS = function() {
    this.calculate = function(package) {

        // calculations...

        return "$45.95";
    }
};

var USPS = function() {
    this.calculate = function(package) {

        // calculations...

        return "$39.40";
    }
};

var Fedex = function() {
    this.calculate = function(package) {

        // calculations...

        return "$43.20";
    }
};

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


function run() {

    var package = { from: "76712", to: "10012", weigth: "lkg" };

    // the 3 strategies

    var ups = new UPS();
    var usps = new USPS();
    var fedex = new Fedex();

    var shipping = new Shipping();
    shipping.setStrategy(ups);
    log.add("UPS Strategy: " + shipping.calculate(package));

    shipping.setStrategy(usps);
    log.add("USPS Strategy: " + shipping.calculate(package));

    shipping.setStrategy(fedex);
    log.add("Fedex Strategy: " + shipping.calculate(package));

    log.show();
}
```

JavaScript Optimized Code

The Namespace pattern is applied to keep the code out of the global namespace. Our namespace is named `Patterns.Classic`. A Revealing Module named `Strategy` returns (i.e. reveals) only a single item: the `Shipping` constructor function which, in turn, only exposes two methods: `setStrategy` and `calculate`.

Here the `setStrategy` takes in a string and assigns the appropriate strategy within the Strategy module. All three shipping strategies, i.e. `UPS`, `USPS`, and `Fedex`, are kept private inside the module's closure. Again, this is a nice example of OO encapsulation and data hiding using JavaScript patterns.

The `Patterns` object contains the `namespace` function which constructs namespaces non-destructively, that is, if a name already exists it won't overwrite it.

The `log` function is a helper which collects and displays results.

```
var Patterns = {
    namespace: function (name) {
        var parts = name.split(".");
        var ns = this;

        for (var i = 0, len = parts.length; i < len; i++) {
            ns[parts[i]] = ns[parts[i]] || {};
            ns = ns[parts[i]];
        }

        return ns;
    }
};

Patterns.namespace("Classic").Strategy = (function () {

    var Shipping = function () {
        var company = "";

        this.setStrategy = function (name) {
            if (name === "UPS") {
                company = new UPS();
            } else if (name === "USPS") {
                company = new USPS();
            } else if (name === "Fedex") {
                company = new Fedex();
            }
        };

        this.calculate = function (package) {
            return company.calculate(package);
        }
    };

    var UPS = function () {
        this.calculate = function (package) {

            // calculations...

            return "$45.95";
        }
    };

    var USPS = function () {
        this.calculate = function (package) {

            // calculations...

            return "$39.40";
        }
    };

    var Fedex = function () {
        this.calculate = function (package) {

            // calculations...

            return "$43.20";
        }
    };
    return { Shipping: Shipping };

})();

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

function run() {

    var shipping = new Patterns.Classic.Strategy.Shipping();

    var package = { from: "76712", to: "10012", weigth: "1kg" };

    // the 3 strategies
    shipping.setStrategy("UPS");
    log.add("UPS Strategy: " + shipping.calculate(package));

    shipping.setStrategy("USPS");
    log.add("USPS Strategy: " + shipping.calculate(package));

    shipping.setStrategy("Fedex");
    log.add("Fedex Strategy: " + shipping.calculate(package));

    log.show();
}
```

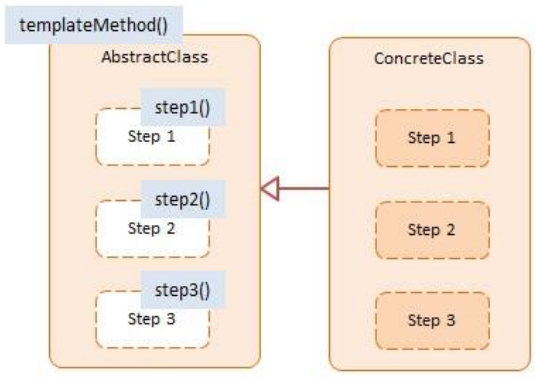
Template Method

The Template Method pattern provides an outline of a series of steps for an algorithm. Objects that implement these steps retain the original structure of the algorithm but have the option to redefine or adjust certain steps. This pattern is designed to offer extensibility to the client developer.

Template Methods are frequently used in general purpose frameworks or libraries that will be used by other developer An example is an object that fires a sequence of events in response to an action, for example a process request. The object generates a 'preprocess' event, a 'process' event and a 'postprocess' event. The developer has the option to adjust the response to immediately before the processing, during the processing and immediately after the processing.

An easy way to think of Template Method is that of an algorithm with holes (see diagram below). It is up to the developer to fill these holes with appropriate functionality for each step.

Diagram



Participants

The objects participating in this pattern are:

- AbstractClass -- In sample code: datastore
    - offers an interface for clients to invoke the templateMethod
    - implements template method which defines the primitive Steps for an algorithm
    - provides the hooks (through method overriding) for a client developer to implement the Steps
- ConcreteClass -- In sample code: mySql
    - implements the primitive Steps as defined in AbstractClass

JavaScript Code

In this example we use JavaScript's prototypal inheritance. The inherit function helps us establish the inheritance relationship by assigning a base object to the prototype of a newly created descendant object.

The `datastore` function represents the AbstractClass and mySql represents the ConcreteClass. `mySql` overrides the 3 template methods: `connect`, `select`, and `disconnect` with datastore-specific implementations.

The template methods allow the client to change datastore (SQL Server, Oracle, etc.) by adjusting only the template methods. The rest, including the order of the steps, stays the same for any datastore.

The `log` function is a helper which collects and displays results.

```
var datastore = {
    process: function() {
        this.connect();
        this.select();
        this.disconnect();
        return true;
    }
};

function inherit(proto) {
    var F = function() { };
    F.prototype = proto;
    return new F();
}

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


function run() {

    var mySql = inherit(datastore);

    // implement template steps

    mySql.connect = function() {
        log.add("MySQL: connect step");
    };
    mySql.select = function() {
        log.add("MySQL: select step");
    };
    mySql.disconnect = function() {
        log.add("MySQL: disconnect step");
    };

    mySql.process();

    log.show();
}
```

JavaScript Optimized Code

The Namespace pattern is applied to keep the code out of the global namespace. Our namespace is named `Patterns.Classic`. A Revealing Module named `Template` returns (i.e. reveals) a single item: the `datastore` object.

A second namespace is created named `Patterns.Utils` which holds utility-type functions. A Revealing Module named `Common` returns (i.e. reveals) two items: `inherit` and our trusted `log` utility. With this we have limited our footprint on the global namespace to a single item, i.e. the `Patterns` root of our namespace.

The `Patterns` object contains the `namespace` function which constructs namespaces non-destructively, that is, if a name already exists it won't overwrite it.

The `log` function is a helper which collects and displays results.

```
var Patterns = {
    namespace: function (name) {
        var parts = name.split(".");
        var ns = this;

        for (var i = 0, len = parts.length; i < len; i++) {
            ns[parts[i]] = ns[parts[i]] || {};
            ns = ns[parts[i]];
        }

        return ns;
    }
};

Patterns.namespace("Classic").Template = (function () {
    var datastore = {
        process: function () {
            this.connect();
            this.select();
            this.disconnect();
            return true;
        }
    };

    return { datastore: datastore };

})();

Patterns.namespace("Utils").Common = (function () {

    var inherit = function (proto) {
        var F = function () { };
        F.prototype = proto;
        return new F();
    };

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

    return {
        inherit: inherit,
        log: log
    };

})();


function run() {

    var utils = Patterns.Utils.Common;

    var store = Patterns.Classic.Template.datastore;
    var mySql = utils.inherit(store);

    // implement template steps

    mySql.connect = function () {
        utils.log.add("MySQL: connect step");
    };
    mySql.select = function () {
        utils.log.add("MySQL: select step");
    };
    mySql.disconnect = function () {
        utils.log.add("MySQL: disconnect step");
    };

    mySql.process();

    utils.log.show();
}
```

Visitor

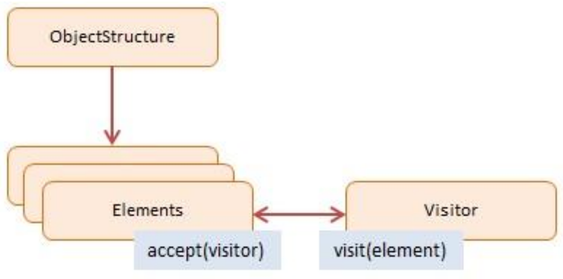
The Visitor pattern defines a new operation to a collection of objects without changing the objects themselves. The new logic resides in a separate object called the Visitor.

Visitors are useful when building extensibility in a library or framework. If the objects in your project provide a 'visit' method that accepts a Visitor object which can make changes to the receiving object then you are providing an easy way for clients to implement future extensions.

In most programming languages the Visitor pattern requires that the original developer anticipates functional adjustments in the future. This is done by including methods that accept a Visitor and let it operate on the original collection of objects.

Visitor is not important to JavaScript because it offers far more flexibility by the ability to add and remove methods at runtime.

Diagram



Participants

The objects participating in this pattern are:

- ObjectStructure -- In sample code: employees array
    - maintains a collection of Elements which can be iterated over
- Elements -- In sample code: Employee objects
    - defines an accept method that accepts visitor objects
    - in the accept method the visitor's visit method is invoked with 'this' as a parameter
- Visitor -- In sample code: ExtraSalary, ExtraVacation
    - implements a visit method. The argument is the Element being visited. This is where the Element's changes are made

JavaScript Code

In this example three employees are created with the `Employee` constructor function. Each is getting a 10% salary raise and 2 more vacation days. Two visitor objects, `ExtraSalary` and `ExtraVacation`, make the necessary changes to the employee objects.

Note that the visitors, in their `visit` methods, access the closure variables `salary` and `vacation` through a public interface. It is the only way because closures are not accessible from the outside. In fact, `salary` and `vacation` are not variables, they are function arguments, but it works because they are also part of the closure.

Notice the `self` variable. It is used to maintain the current context (stored as a closure variable) and is used in the `visit` method.

The `log` function is a helper which collects and displays results.

```
var Employee = function (name, salary, vacation) {

    var self = this;

    this.accept = function (visitor) {
        visitor.visit(self);
    };
    this.getName = function () {
        return name;
    };
    this.getSalary = function () {
        return salary;
    };
    this.setSalary = function (sal) {
        salary = sal;
    };
    this.getVacation = function () {
        return vacation;
    };
    this.setVacation = function (vac) {
        vacation = vac;
    };
};

var ExtraSalary = function () {
    this.visit = function (emp) {
        emp.setSalary(emp.getSalary() * 1.1);
    };
};
var ExtraVacation = function () {
    this.visit = function (emp) {
        emp.setVacation(emp.getVacation() + 2);
    };
};

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


function run() {

    var employees = [
        new Employee("John", 10000, 10),
        new Employee("Mary", 20000, 21),
        new Employee("Boss", 250000, 51)
    ];

    var visitorSalary = new ExtraSalary();
    var visitorVacation = new ExtraVacation();

    for (var i = 0, len = employees.length; i < len; i++) {

        var emp = employees[i];

        emp.accept(visitorSalary);
        emp.accept(visitorVacation);
        log.add(emp.getName() + ": $" + emp.getSalary() +
            " and " + emp.getVacation() + " vacation days");
    }

    log.show();
}
```

JavaScript Optimized Code

The Namespace pattern is applied to keep the code out of the global namespace. Our namespace is named `Patterns.Classic`. A Revealing Module named `Visitor` returns (i.e. reveals) only a single item: the `Employee` constructor function.

In the `run` method an array with 3 employees is created. Next, two 'visitor' functions are defined: `extraSalary` and `extraVacation` which are going to be applied to each employee. We could have added these to the Visitor module, but this better demonstrates that you can arbitrarily create and apply functions to any object. The only requirement is that the properties and methods referenced in the function do exist on the object.

This shows that the Visitor pattern is essentially native to JavaScript as expressed by the Apply Invocation pattern.

The `Patterns` object contains the `namespace` function which constructs namespaces non-destructively, that is, if a name already exists it won't overwrite it.

The `log` function is a helper which collects and displays results.

```
var Patterns = {
    namespace: function (name) {
        var parts = name.split(".");
        var ns = this;

        for (var i = 0, len = parts.length; i < len; i++) {
            ns[parts[i]] = ns[parts[i]] || {};
            ns = ns[parts[i]];
        }

        return ns;
    }
};

Patterns.namespace("Classic").Visitor = (function () {

    var Employee = function (name, salary, vacation) {

        this.getName = function () { return name; };
        this.setName = function (value) { name = value; };

        this.getSalary = function () { return salary; };
        this.setSalary = function (value) { salary = value; };

        this.getVacation = function () { return vacation; };
        this.setVacation = function (value) { vacation = value; }
    };

    return { Employee: Employee };

})();

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


function run() {

    var visitor = Patterns.Classic.Visitor;

    var employees = [
        new visitor.Employee("John", 10000, 10),
        new visitor.Employee("Mary", 20000, 21),
        new visitor.Employee("Boss", 250000, 51)
    ];

    // create 'visitor' functions
    var extraSalary = function () {
        this.setSalary(this.getSalary() * 1.1)
    };

    var extraVacation = function () {
        this.setVacation(this.getVacation() + 2)
    };

    for (var i = 0, len = employees.length; i < len; i++) {

        var emp = employees[i];

        // apply 'visitor' functions
        extraSalary.apply(emp);
        extraVacation.apply(emp);

        log.add(emp.getName() + ": $" + emp.getSalary() +
            " and " + emp.getVacation() + " vacation days");
    }

    log.show();
}
```