**Software Design Patterns**

**Definition:**

A design pattern is a general reusable solution to a commonly occurring problem within a given context in software design. A design pattern is not a finished design that can be transformed directly into source or machine code. It is a description or template for how to solve a problem that can be used in many different situations.

 Design patterns are proven ways of programming to help make your code more maintainable, scalable, and decoupled, all of which are necessary when creating large JavaScript applications, especially in large groups.

**Types of Design Patterns**

In software development, design patterns are generally grouped into a few categories. They are explained in brief below:

1. **Creational** patterns focus on ways to create objects or classes. This may sound simple (and it is in some cases), but large applications need to control the object creation process.
2. **Structural** design patterns focus on ways to manage relationships between objects so that your application is architected in a scalable way. A key aspect of structural patterns is to ensure that a change in one part of your application does not affect all other parts.
3. **Behavioral** patterns focus on communication between objects.
4. **Creational Pattern:**

* **Singleton Pattern:**
* It restricts the instantiation of a class to a single object.
* In server side languages the singleton pattern is generally used to handle a connection to a database because it is just a waste of resources to create more than one database connection for each request.
* If a class has the same functionality every time it is preferred to use singleton.
* Besides this, in Javascript, singleton pattern allows you to namespace objects and functions to keep them organized.
* **Prototype Pattern:**
* It is a pattern in which objects are created based on the template on the existing objects through cloning.
* For instance

var Person={

numfeet : 2,

numheads:1,

};

var a=Object.create(Person);

document.write(a.numheads);

* The properties and methods in the Person object gets applied to the prototype of (a) object.
* The goal is to minimize the amount the amount of work needed in creating new objects when the initialization routines are expensive. It is like we create a template once and then all the objects we create afterwards use the template the starting point to avoid repeating the work again.

1. **Structural Pattern:**

* **Decorator Pattern:**
* If we want to add or change the functionality on a class without creating a subclass for every combination of functionality.
* For instance, we have a car with default functionalities. The car has several options that we can add on to the car(eg. Power locks, power windows and so on).
* It wraps the base object with the decorator object that has the same interface as the base object. The decorator has few options how it can handle the objects:
  1. If can completely override the object which it wraps.
  2. If the decorator doesn’t affect the behavior of the method, it can just pass through the wrapped object.

1. **Behavioral Pattern:**

* **Chain of Responsibility Pattern:**
* There are three parts to the chain of responsibility: sender , receiver and request. The sender makes the request, the receiver is the chain of one or more objects that choose whether to handle the requests or pass it on.
* The sender only knows about the first object and nothing about the first receiver.
* It is nothing but just passing the request from one object to other until the request is fulfilled.
* For instance, you can pass the mortgage application requests to a bank manager, and if the manager doesn’t approve the loan, it can be passed to his supervisor and so on.

**Example(Prototype Pattern, Singleton Pattern, Chain of responsibility Pattern)**

<html>

<head>

<script>

var Chainable = function () {

this.nNumber = 0;

};

Chainable.prototype.add = function (nNumber) {

this.nNumber += nNumber;

return this;

};

Chainable.prototype.multiply = function (nNumber) {

this.nNumber \*= nNumber;

return this;

};

Chainable.prototype.toString = function()

{

return this.nNumber.toString();

};

var oChainable = new Chainable();

document.write( oChainable.add(8).add(4).multiply(30).toString());

</script>

</head>

</html>

**Explanation:**

* The above example consist of singleton design since we are creating on one instance of the class to access the methods.
* We have used prototype pattern since the methods used are almost the same as each and every method consist of one argument. As per the definition we have defined the function and then use this function to avoid the repetition.
* The above example also uses the chain of responsibility pattern. When we are printing the output we are calling all the function in the chain. One function does the calculation and sends the results to other function.
* Based on the requirements we can use different design patterns interchangeably.

Example output:

<http://web.njit.edu/~np298/designpattern/example.html>