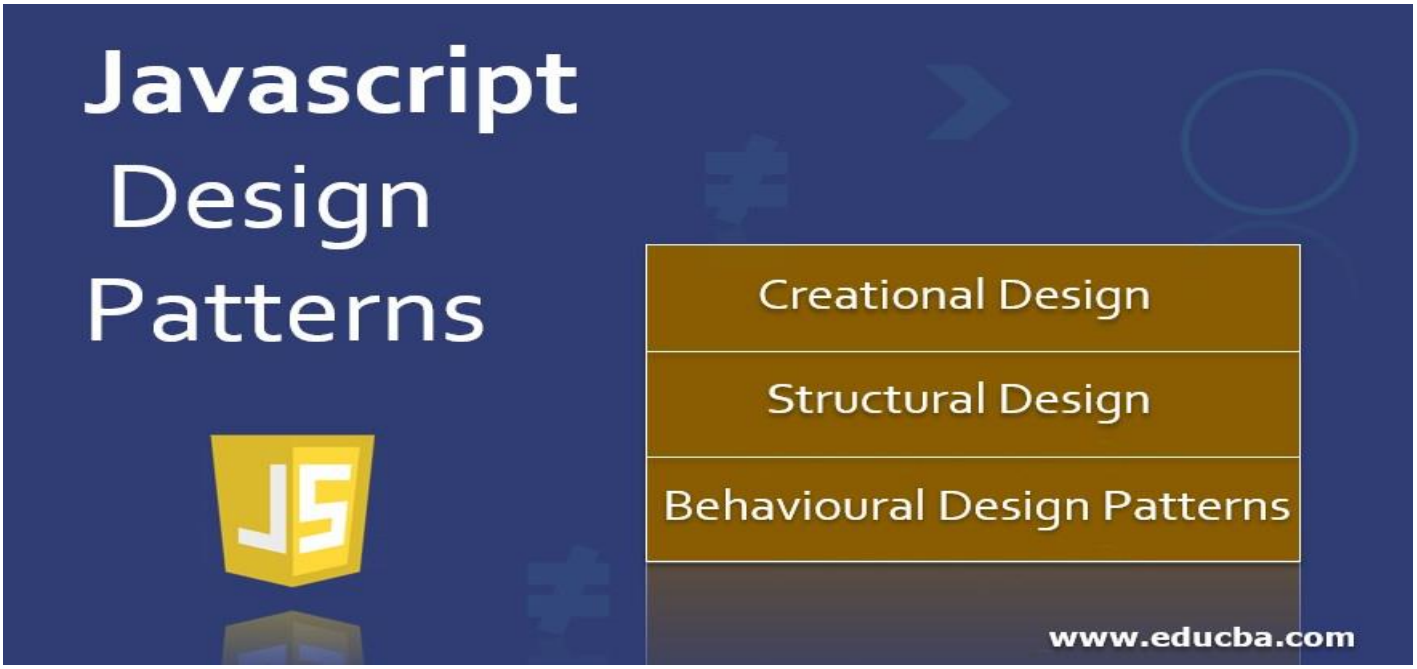
JavaScript Design Pattern



Eng.Nadia Saleh

Design Pattern is a widely acknowledged concept in the software engineering industry in terms of the benefits it brings to areas of code-reuse and maintainability. As a software developer, you likely stumble upon this term at one point. Unsurprisingly, without even knowing it, the chances are that you might have already implemented them somewhere in the development journey.

A **design pattern** is used to identify reusable solutions that can be applied to recurring problems that software developers commonly face during software design. They represent time-tested solutions and best practices adopted by object-oriented software developers over time.

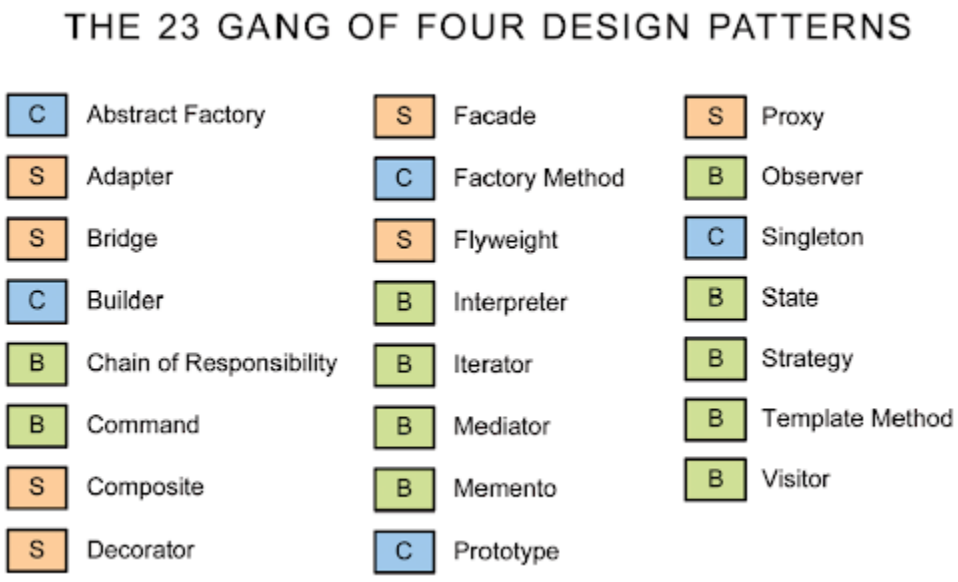
This blog will be your guide to everything you need to know about popular JavaScript design patterns. The only prerequisite is that you should have basic knowledge of JavaScript and Object-Oriented Programming concepts.

Starting with the historical perspective, we will do an in-depth exploration of various common JavaScript design patterns from an object-oriented view. By the end, you will be accustomed to various JavaScript design patterns along with a basic idea of their implementation.

History of Design Pattern

Since its inception, the concept of design pattern has been around in the programming world. But it was not formalized till 1994 when one of the most influential work was published called “Design Patterns: Elements Of Reusable Object-Oriented Software” – written by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides – a group that became known as the Gang of Four (or GoF).

In this book, 23 object-oriented design patterns are featured. Since then, the “pattern approach” became popular in the software engineering industry, and after that, dozens of other patterns have been discovered.



What is a Design Pattern?

Design patterns can be considered pre-made blueprint applied to solve a particular design problem. It is not a finished piece of code that can be directly applied to your program. But rather, it is more like a template or description that can give you an idea of approaching a problem and inspiring solutions. Hence, the code for the two separate programming scenarios, implementing the same pattern can be different.

Now, if you are wondering how a pattern gets discovered, it's simple. When the same solution gets repeated over and over, someone will eventually recognize it, put a name to it, and then describe the solution in detail. That's how a pattern gets discovered. Definitely, they were not forged overnight.

A design pattern is often confused with algorithms.

Why Patterns?

As previously mentioned, we already use patterns every day. They help us solve recurring design problems. But is it necessary to spend time learning them? Let's look into a few key benefits that design patterns grant us.

1. Avoid reinventing the wheel:

Most of the frequently faced design problems already have a well-defined solution that is associated with a pattern. Patterns are proven solutions that can speed up development.

2. Codebase Maintenance:

Patterns help in implementing DRY(Do not Repeat Yourself) – the concept which helps to prevent your codebase from growing large and unwieldy.

3. Easily reused:

Reusing patterns assists in preventing minor subtle issues that can cause major problems in the application development process. This also improves code readability for coders and architects familiar with the patterns.

4. Enables efficient communication:

Patterns add to a developer's vocabulary. This allows developers to communicate using well-known, well-understood names for software interactions, making communication faster.

5. Improve your object-oriented skills:

Now even if you never encounter any of these problems, learning patterns can give you insights into various approaches to solving problems using object-oriented principles.

Criticism of Patterns

Over time design patterns have also received a fair share of criticism. Let's peek into the popular arguments against patterns.

1. Increases Complexity:

Inappropriate use of patterns creates undesired complexity. This is a problem suffered by many novices, who try to apply the pattern wherever they can think of, even in situations where simpler code would do just fine.

2. Reduced Relevance:

In "*Design Patterns in Dynamic Languages*," Peter Norvig points out that over half of the design patterns in the 1994 book (written by GoF) are workarounds for missing language features. In many cases, patterns just become kludges that gave the programming language the much-needed super-abilities it lacked then.

As the language features, frameworks, and libraries evolved, there is no reason to use a few patterns anymore.

3. Lazy Design:

As suggested by Paul Graham in "Revenge of the Nerds" (2002), patterns are a form of lazy design, when the developer is not focused on the problem requirement at hand. Instead of creating a new and appropriate design for the problem, they might just reuse the existing design patterns because they think they should.

So far, we have seen what design patterns are and also discussed their advantages and disadvantages. Now it's time for in-depth exploration of various types of JS design patterns available.

JavaScript Design Patterns

JavaScript is one of the most in-demand programming languages for web development today. As we will be concentrating on JavaScript design patterns in this article, let's just have a quick recap of essential JavaScript features that will aid in smoother understanding.

a) Flexible with programming styles
JavaScript has support for procedural, object-oriented, and functional programming styles.

b) Supports First-class Functions
This means functions can be passed as arguments to other functions just like a variable.

c) Prototype-based Inheritance
Though JavaScript supports objects, unlike other OOPs languages, JavaScript doesn't have the concept of class or class-based inheritance in its basic form. Instead, it uses something called prototype-based or instance-based inheritance.

Categories of Design Pattern

Based on intent, the JavaScript design pattern can be categorized into 3 major groups:

a) Creational Design Pattern

These patterns focus on handling object creation mechanisms. A basic object creation approach in a program can lead to an added complexity. Creational JS design patterns aim to solve this problem by controlling the creation process.

Few patterns that fall under this category are – Constructor, Factory, Prototype, Singleton, etc.

b) Structural Design Patterns

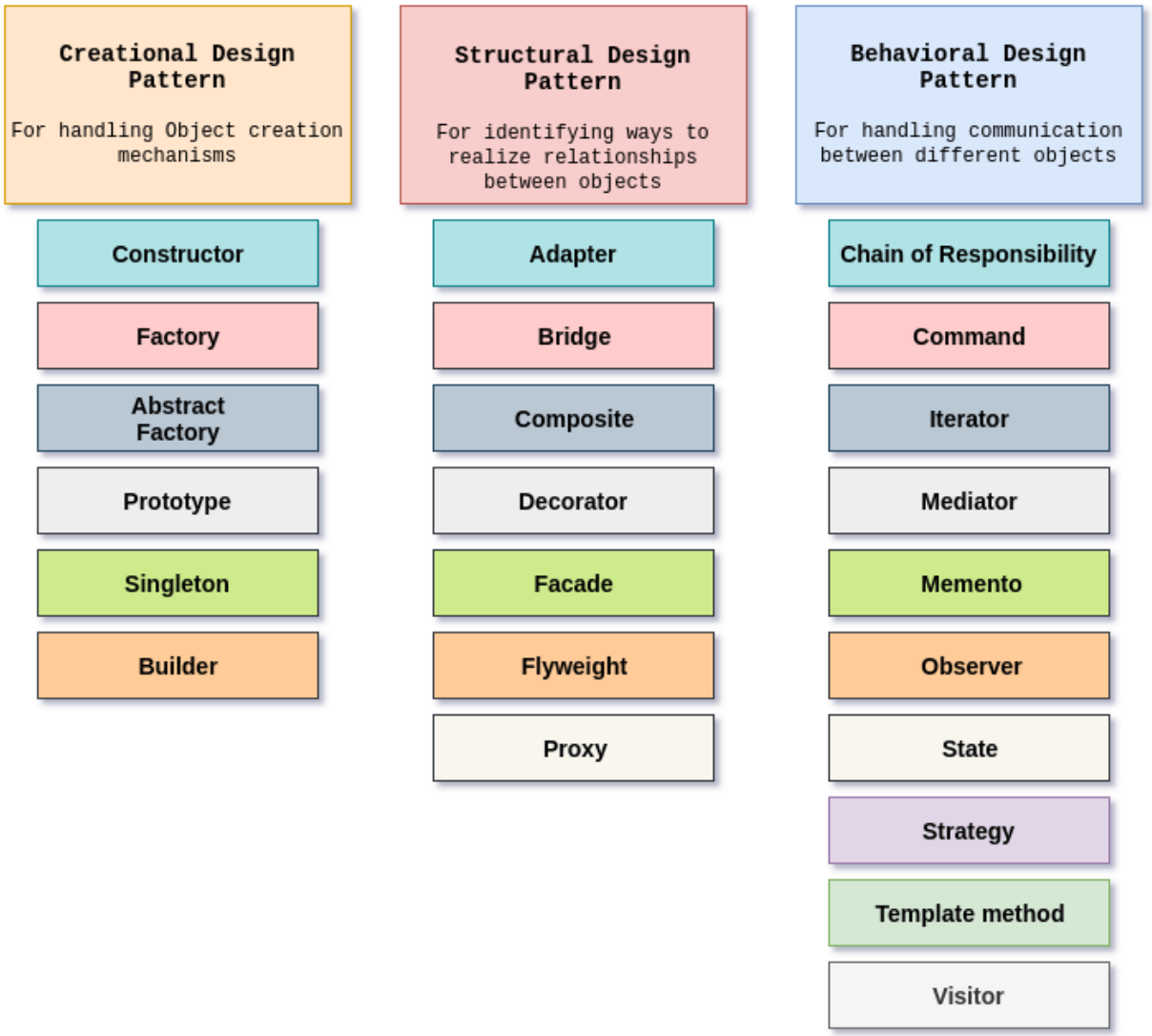
These patterns are concerned with object composition. They explain simple ways to assemble objects and classes into larger structures. They help ensure that when one part of a system changes, the entire structure of the system doesn't need to do the same, keeping them flexible and efficient.

Few patterns that fall under this category are – Module, Decorator, Facade, Adapter, Proxy, etc.

c) Behavioral Design Patterns

These patterns focus on improving the communication and assignment of responsibilities between dissimilar objects in a system.

Few patterns that fall under this category are – Chain of Responsibility, Command, Observer, Iterator, Strategy, Template, etc.



Creational Design Patterns

1. Constructor Pattern

The constructor pattern is one of the most simple, popular, and modern JS design patterns. As suggested by the name, the purpose of this pattern is to aid constructor creation.

In Addy's words-

"A constructor is a special method used to initialize a newly created object once the memory has been allocated for it. In JavaScript, as almost everything is an object, we're most often interested in object constructors."

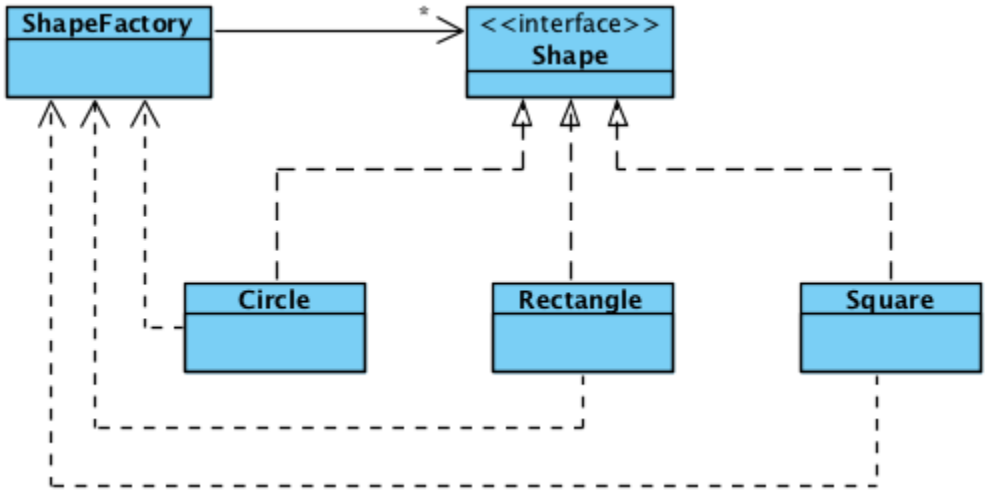
```
function Person(name,age) {
    this.name = name;
    this.age = age;
    this.getDetails = function () {
        console.log(`${this.name} is ${this.age} years old!`);

}
}

//  b) ES6 "class" syntax

class Person {
constructor(name, age) {
    this.name = name;
    this.age = age;
    this.getDetails = function () {
        console.log(`${this.name} is ${this.age} years old!`);
    };
}
}

//Creating new instance of Person
const personOne = new Person('Nadia',29);
personOne.getDetails(); // Output - "Nadia is 29years old!
```

2. Factory Pattern

The Factory pattern is another creational pattern concerned with creating objects but using some sort of generic interface. According to GoF's book, this pattern has the following responsibility.

"Define an interface for creating an object, but let subclasses decide which class to instantiate."

This pattern is typically used when we need to handle object groups that share similar characters yet are different through appropriate custom calls. An example would bring more clarity.



Note: Though the definition particularly mentions that an interface needs to be defined, we don't have interfaces in JavaScript. Therefore, we are going to implement it using an alternative way.

Example:

Here, the **shapeFactory** constructor is responsible for creating new objects of the constructors' Rectangle, Square, and Circle. The **createShape()** inside **shapeFactory** takes in parameters, depending on which it delegates the responsibility of object instantiation to the respective class.

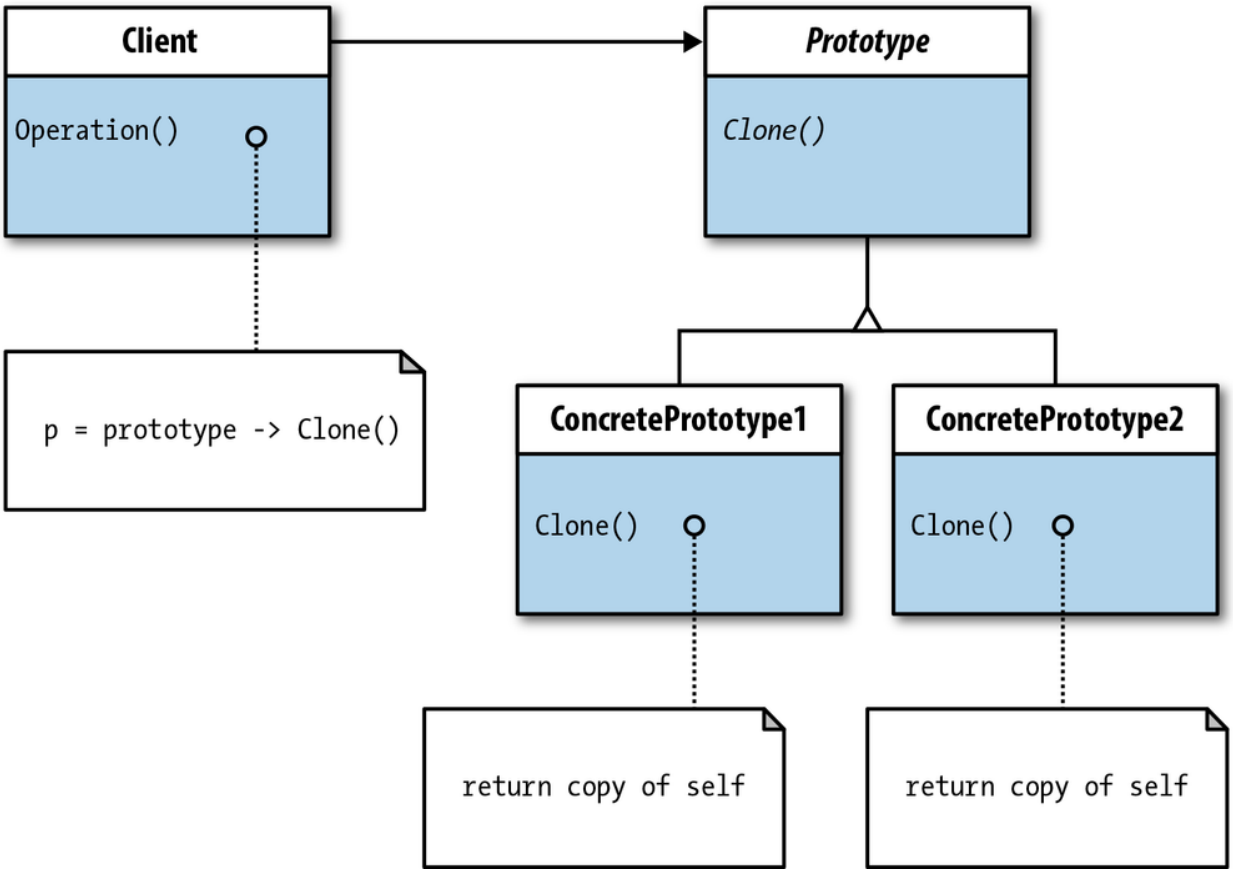
```
//Factory method for creating new shape instances
function shapeFactory(){
    this.createShape = function (shapeType) {

        var shape;
        switch(shapeType){
            case "rectangle":
                shape = new Rectangle();
                break;
            case "square":
                shape = new Square();
                break;
            case "circle":
```

```
                shape = new Circle();
                break;
            default:
                shape = new Rectangle();
                break;
        }
        return shape;
    }
}

// Constructor for defining new Rectangle
var Rectangle = function () {
    this.draw = function () {
        console.log('This is a Rectangle');
    }
};

// Constructor for defining new Square
var Square = function () {
    this.draw = function () {
        console.log('This is a Square');
    }
};

// Constructor for defining new Circle
var Circle= function () {
    this.draw = function () {
        console.log('This is a Circle');
    }
};

var factory = new shapeFactory();
//Creating instance of factory that makes rectangle,square,circle respectively
var rectangle = factory.createShape('rectangle');
var square = factory.createShape('square');
var circle= factory.createShape('circle');

rectangle.draw();
square.draw();
circle.draw();
```

3. Prototype Pattern

An object that supports cloning is called a **prototype**. Using the prototype pattern, we can instantiate new objects based on a template of an existing object through cloning.

As the prototype pattern is based on prototypal inheritance, we can utilize the native prototypical strengths of JavaScript. In the previous JS design patterns, we were trying to imitate features of other languages in JavaScript, which is not the case here.



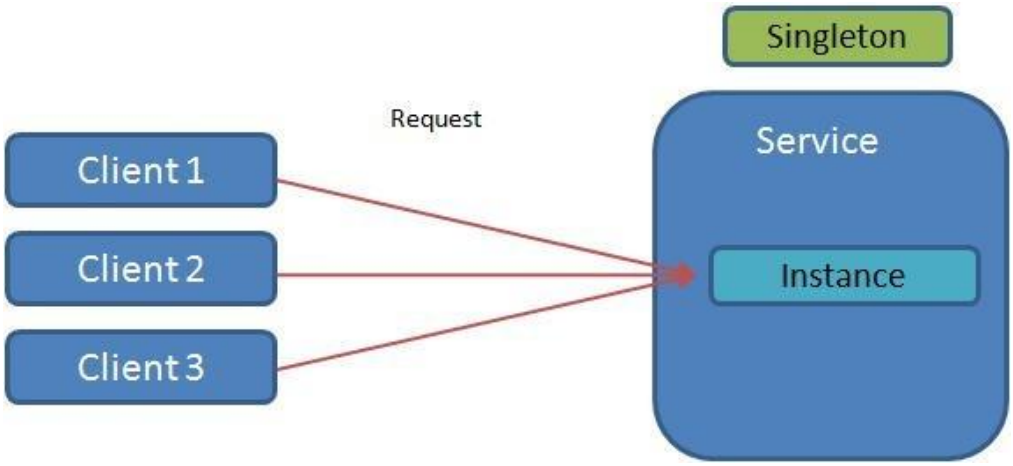

```
const car = {
    noOfWheels: 4,
    start() {
      return 'started';
    },
    stop() {
      return 'stopped';
    },
  };
//using Object.create to create clones - as recommended by ES5 standard
  const myCar = Object.create(car, { owner: { value: 'John' } });

  console.log(myCar.__proto__ === car); // true
```

4. Singleton Pattern

The singleton pattern is a creational JavaScript design pattern that restricts the instantiation of a class to a single object. It creates a new instance of the class if one doesn't exist and if existing already, it simply returns a reference to it. It is also known as the Strict Pattern.

A singleton pattern solves two problems at the same time, violating the Single Responsibility Principle.

- Guarantees that there is only a single instance of a class.
- Provide a global access point to this instance.




```
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));
}
```

Structural Design Patterns

1. Adapter Pattern

The adapter is a structural JS design pattern that allows objects or classes with incompatible interfaces to collaborate. It matches interfaces of different classes or objects; therefore, they can work together despite incompatible interfaces. It is also referred to as the Wrapper pattern.

A real-world analogy would be trying to connect a projector to a laptop. The projector might have a VGA plug, and the laptop might have an HDMI plug. So we require an adapter that can make these two unrelated interfaces compatible.

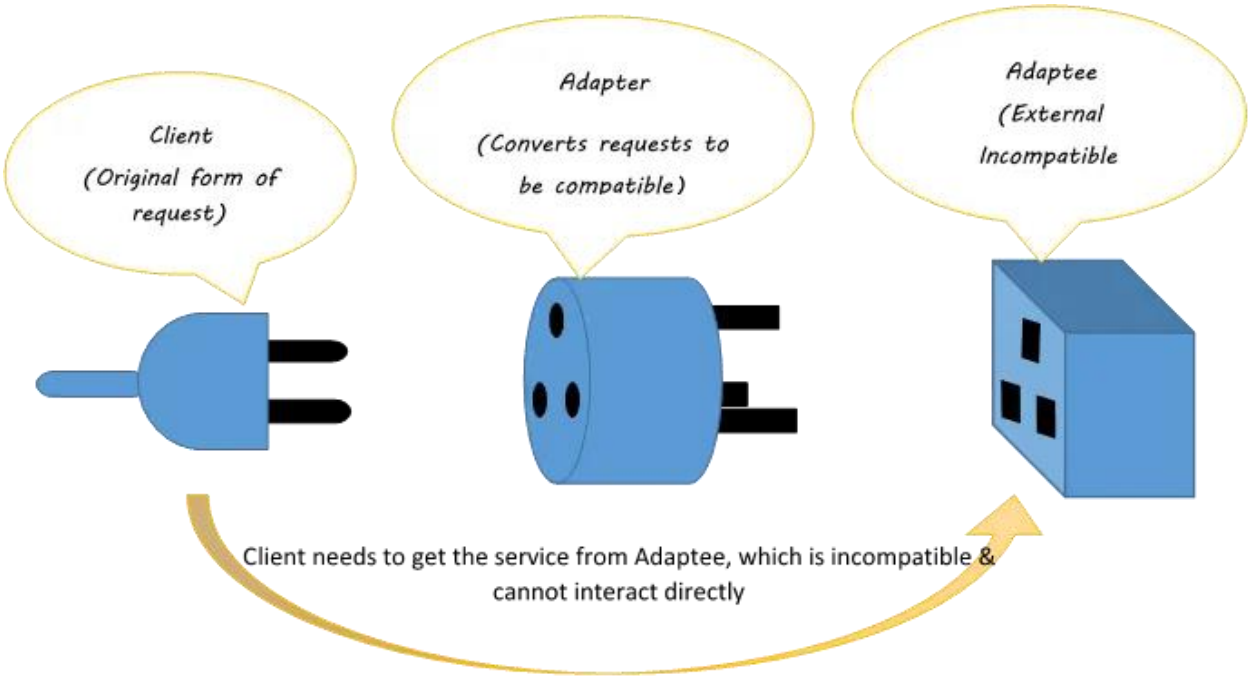
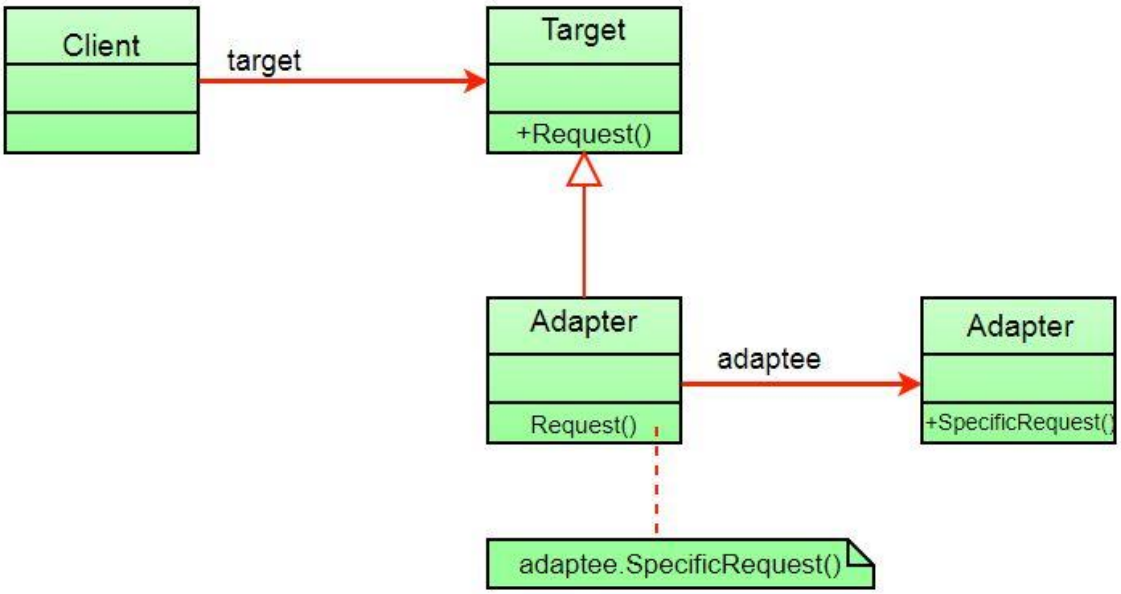


Figure 1-Adapter Pattern Concept

Adapter UML:

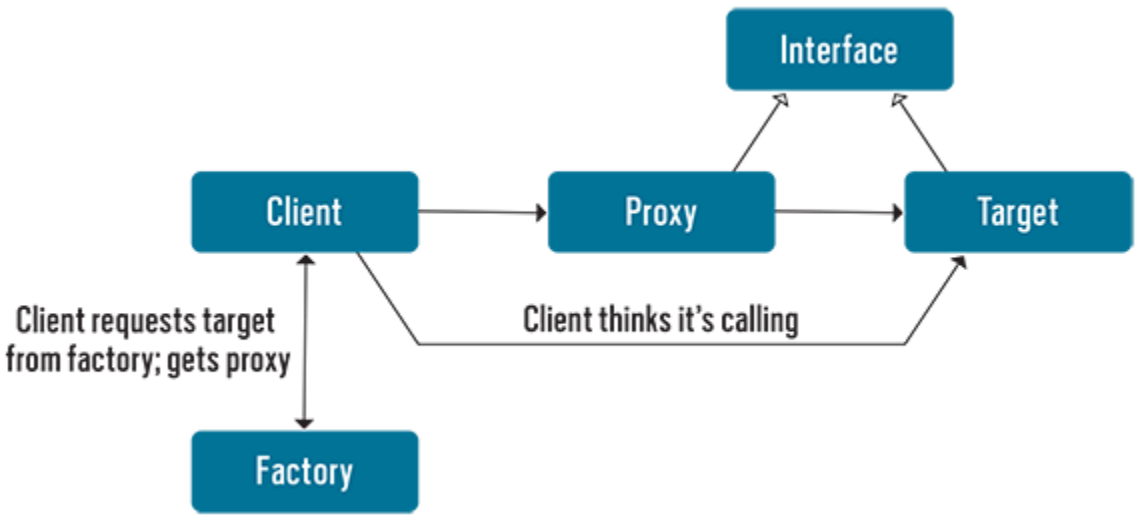



```
// old interface
function TicketPrice() {
    this.request = function(start, end, overweightLuggage) {
        // price calculation code...
        return "$150.34";
    }
}

// new interface
function NewTicketPrice() {
    this.login = function(credentials) { /* process credentials */ };
    this.setStart = function(start) { /* set start point */ };
    this.setDestination = function(destination) { /* set destination */ };
    this.calculate = function(overweightLuggage) {
        //price calculation code...
        return "$120.20";
    };
}

// adapter interface
function TicketAdapter(credentials) {
    var pricing = new NewTicketPrice();

    pricing.login(credentials);
```

```
    return {
        request: function(start, end, overweightLuggage) {
            pricing.setStart(start);
            pricing.setDestination(end);
            return pricing.calculate(overweightLuggage);
        }
    };
}

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

// original ticket pricing and interface
var price = pricing.request("Bern", "London", 20);
console.log("Old price: " + price);

// new ticket pricing with adapted interface
price = adapter.request("Bern", "London", 20);
console.log("New price: " + price);
```

2-Proxy Pattern

As the name suggests, the Proxy Pattern provides a surrogate or placeholder for another object to control access, reduce cost, and reduce complexity. The proxy could interface to anything – a network connection, a large object in memory, a file, or some other resource that is expensive or impossible to duplicate.



Here, we will create a proxy object that ‘stands in’ for the original object. The proxy interface will be the same as that of the original object so that the client may not even be aware they are dealing with a proxy rather than the real object. In the proxy, extra functionality can be provided, for example, caching, checking some preconditions, etc.

There are three common situations in which the Proxy pattern is applicable.

1. A **virtual proxy** is a placeholder for expensive to create or resource-intensive objects.
2. A **remote proxy** controls access to the remote object.
3. A **protective proxy** controls access rights to a sensitive master object. The caller’s access permissions are checked prior to forwarding the request.

Example:

The following code will aid you in getting a gist of Proxy implementation. We have an external API FlightListAPI for accessing Flight Details databases. We will create a proxy FlightListProxy which will act as the interface through which the client can access the API.

```
var FlightListAPI = function() {
    //creation
    };

    FlightListAPI.prototype = {
    getFlight: function() {
        // get master list of flights
        console.log('Generating flight List');
    },

    searchFlight: function(flightDetails) {
        // search through the flight list based on criteria
        console.log('Searching for flight');
    },

    addFlight: function(flightData) {
        // add a new flight to the database
        console.log('Adding new flight to DB');
    }
    };

    // creating the proxy
    var FlightListProxy = function() {
        // getting a reference to the original object
    this.flightList = new FlightListAPI();
    };

    FlightListProxy.prototype = {
    getFlight: function() {
```

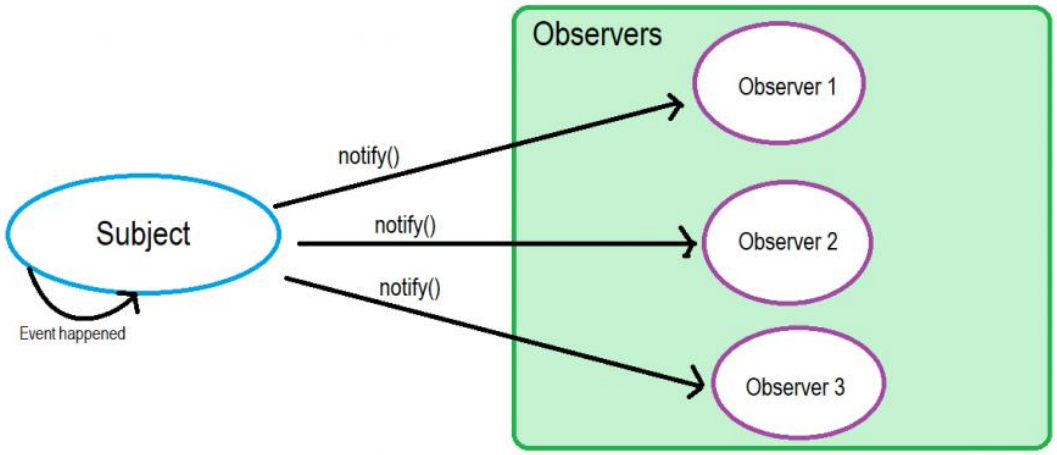
```
        return this.flightList.getFlight();
    },

    searchFlight: function(flightDetails) {
        return this.flightList.searchFlight(flightDetails);
    },

    addFlight: function(flightData) {
        return this.flightList.addFlight(flightData);
    },

    };

    console.log("----------With Proxy----------")
    const proxy = new FlightListProxy()
    console.log(proxy.getFlight());
    /*

    OUTPUT

    ----------With Proxy----------
    Generating flight List


    */
```

Behavioural Design Pattern

Observer Pattern

The Observer is a behavioral JS design pattern that lets you define a subscription mechanism to notify multiple objects (observers) about any events that happen to the object (subject) they're observing. This pattern is also called Pub/Sub, short for Publication/Subscription. It defines a one-to-many dependency between objects, promotes loose coupling, and facilitates good object-oriented design.

The observer pattern is the foundation of event-driven programming. We write event handler functions that will be notified when a certain event fires.



Example:

We have set up a Subject function Click and extended it using the prototype. We have created methods to subscribe and unsubscribe objects to the Observer collection, which is handled by the clickHandler function. Also, there is a fire method to propagate any changes in the Subject class object to the subscribed Observers.

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

Click.prototype = {

    subscribe: function(fn) {
        this.observers.push(fn);
    },

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

    fire: function(o, thisObj) {
        var scope = thisObj;
        this.observers.forEach(function(item) {
            item.call(scope, o);
        });
    }
```

```
}

function run() {

    var clickHandler = function(item) {
        console.log("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');

}

/* OUTPUT:

Fired:event #1
Fired:event #3

*/
```