Design Patterns in JavaScript

Learn about common design patterns in JavaScript.

As applications become larger, thoughtful code composition gains importance. As you add moving parts, it can be difficult to maintain, read, or reuse code if the relationships between those different parts are not designed with intention. Without thoughtful codebase design, you might spend more time fixing bugs and updating duplicate code rather than writing new features.

Luckily, a lot of common problems you'll encounter have already been solved in the form of *design patterns*. Design patterns are systems for solutions, rather than code you can directly use. Patterns give names and rules for solving types of problems, and the solution is then an implementation of objects and classes to solve the problem. There are even built-in objects in JavaScript that help with implementation of some common design patterns.

Overall, solutions that implement design patterns tend to be:

- DRY (Don't repeat yourself)
- Modular
- Reusable
- Easier to maintain
- Easier to discuss with peers at a high level (vs diving into implementation details)

Watch Out for Anti-Patterns

In contrast to design patterns being best practices, anti-patterns describe ineffective solutions to problems that result in bad things happening, such as:

- Namespace pollution — unexpected behavior from interactions from different clients
- Increased code complexity
- Code being difficult to understand and update
- Difficulty with testing and debugging code

Luckily, anti-patterns are well documented to help developers from making the same mistakes that have been made in the past.

You'll notice some "approved" design patterns also make the list as anti-patterns. In those cases, if the end result winds up being a good or bad choice depends on the implementation context. Because of this, it's important to assess your solution and how it will impact your code base when you are making design decisions. We'll take a look at how to select the right design pattern later in the article.

Some common anti-patterns include:

- *God Object*: This name is given to an object that does or knows too much. It is an all-knowing and all-encompassing object. While it might seem easier at first to have all of your properties and methods in one place, doing so can cause issues down the line when you go to update and maintain the code base or collaborate with other developers.
- *Big Ball of Mud/ Spaghetti Code*: This is when your code lacks any perceivable architecture. It is hard to figure out where code for a specific functionality is located and what other code depends on it.
- *Yo-Yo Problem*: If you find yourself moving from class definition to class definition to understand inheritance and what is happening among classes, you are experiencing the yo-yo problem, which is named due to your head yo-yoing across the screen.
- *Singleton*: We'll explore this below as a classic design pattern, but it's worth noting that if used improperly, the Singleton pattern can take you into anti-pattern territory because of the restrictions it imposes.
- Polluting the global namespace (i.e. defining too many variables at the global scope level).
- Modifying or extending the `Object` class prototype. All objects in JavaScripts have a `Prototype` property that can be altered with methods and properties and all new objects inherit from this root `Object` by default. However, altering it is a huge no-no.

What an Anti-Pattern Looks Like

In the example below of what not to do, `Customer` objects are all-knowing and have too much responsibility–this code follows the "God Object" anti-pattern.

Let's look at what is going on in the `Customer` class that is not working well. This class creates a `Customer` object with properties about a customer and their transactions.

```
class Customer {
    constructor(userId, first, last) {
        this.userId = userId;
        this.first = first;
        this.last = last;
        this.itemsPurchased = [];
        this.transactions = [
            { transactionId: 1, transactionTotal: 433 },
            { transactionId: 2, transactionTotal: 289 },
            { transactionId: 3, transactionTotal: 99 },
            { transactionId: 4, transactionTotal: 600 }
        ];
 }
    getTotal() {
        return this.transactions.reduce((a,b) => ({transactionTotal:
a.transactionTotal + b.transactionTotal}), 0).transactionTotal;
 }
}

let bilbo = new Customer(1, "Bilbo", "Baggins");
bilbo.getTotal(); // 1421
```

We'd want to refactor the above code so the `Customer` objects are only responsible for information directly about the customer, like the customer id, and first and last names. New types of classes — like Transactions, Transaction, and Products — can handle additional information and methods.

Now that we know all about what not to do, let's learn more about the patterns that you should be using. Design patterns are commonly grouped into a classification scheme that has three purpose-based categories:

- Creational
- Structural
- Behavioral

Let's take a look at each category and check out some common design patterns in action.

The Creational Category

When you need to control how your objects are created or instantiated, *Creational* patterns are the right fit.

Creational patterns include:

- Factory
- Singleton
- Abstract Factory
- Constructor
- Prototype

Let's take a look at two Creational design patterns: the Factory and Singleton patterns.

Factory Pattern

Imagine that you want to keep track of your reading list — each book would have a title, author, number of pages, year written, reading status, and more. If you needed to create an object for each book, you would have to write many lines of repetitive code.

To speed up the process of instantiating your reading list, you can make use of the *Factory Pattern*. Functions that use the Factory Pattern use a predefined template to return an object with properties and methods. The arguments are used to construct the object, while the methods are usually part of the template.

Implementation of the Factory Pattern

In the code example below, the function `createBook()` takes 3 arguments: `title`, `author`, and `read`. (The third argument is optional.) The function returns an object literal with 3 properties (`title`, `author`, and `read`) and 2 methods (`.getDescription()` and `.readBook()`).

```
function createBook(title, author, read = false) {
    return {
        title: title,
        author: author,
        read: read,
        getDescription() {
            console.log(`${this.title} was written by ${this.author}.
I ${this.read ? "have" : "have not"} read it.`);
        },
        readBook() {
            this.read = true;
        },
    }
}
```

We can then instantiate objects and call the methods that become part of the object's properties, or modify the properties directly.

```
const beloved = createBook("Beloved", "Toni Morrison");
console.log(beloved);
/*
{
    title: 'Beloved',
    author: 'Toni Morrison',
    read: false,
    getDescription: [Function: getDescription],
    readBook: [function: readBook]
}
*/

// call the `.readBook()` method
beloved.readBook(); // read is updated to true

// modifies the property directly
beloved.title = "I can change the property."
```

Singleton Pattern

As the name implies, we use the *Singleton design pattern* when we need exactly one instance of a class. Often, it's used with the goal of managing global application state, but without using actual global scope. Singletons act as a shared resource namespace with a single point of access for functions. You might be wondering when you'd only want a single instance of an object. Here are a few real-world use cases:

- Thread pools
- Caches
- Logging options
- Configuration settings
- Browser load time impact of Singleton globally accessible variables vs real global variables (that was a mouth full)
- Database connections: reuse existing connections instead of creating new ones, which would increase application and database load

Note: Many of those options can also be handled with modules.

The single instance restriction is established through the way the class is implemented. A method can be written to check if an instance already exists, and only return a new object if it doesn't already exist.

Implementation of the Singleton Pattern

In the code snippet, we implement a Singleton class called `Singleton`. The `constructor()` is critical to the implementation. In the code, we check if an `instance` property already exists. If not, we set that property. If it does, we return the `instance` property. There is a method named `.getInstance()` defined as well — this is not necessary but makes your code easier to reason with.

When you instantiate your instance of `Singleton`, you use the `new` operator when calling `Singleton()`. Given the code in the `constructor()` method, you could theoretically always call your single instance as `new Singleton()` because it will always return your original instance. However, by providing a `.getInstance()` method, you can instead call it as `Singleton.getInstance()`, which is easier to understand and clearer in intent.

```
class Singleton {
    constructor() {
        if (!Singleton.instance) {
            Singleton.instance = this;
        }
        return Singleton.instance;
    }
    static getInstance() {
        return this.instance;
    }
}

const instance = new Singleton();
console.log(Singleton.getInstance()); // logs "Singleton {}"
```

The Structural Category

Structural design patterns handle relationships between objects. They define how objects and classes can be composed to provide new functionality to objects or create larger structures. For example, an object can be used in the definition of another object to make new functions available, or classes can inherit from super classes.

Structural patterns include:

- Facade
- Proxy
- Flyweight
- Adapter
- Decorator
- Composite
- Bridge

Proxy Pattern

The *Proxy Pattern* protects access to an object by acting as a placeholder that intercepts and redefines the operations of the target object. This pattern is particularly useful for things like network requests, as it can help avoid redundant requests.

There is a `Proxy` object built into ES6 that can be used to implement the proxy pattern. This object has two parameters:

- `target`: the object that is being proxied
- `handler`: a definition of any custom behavior handled by the proxy object

If you use the `Proxy()` constructor with an empty handler, it would just behave like the target object. `Proxy` objects have built-in handler function objects, which are called *traps*. Traps are used to call the target object.

`Proxy` objects are used alongside the `Reflect` object, which has methods with the exact same name as the `Proxy` object's traps. The difference is the `Reflect` methods forward default operations to the target object.

In the code example below, the `get()` trap is used to modify property access of the target object. On the other hand, `Reflect.get()` is a static method that retrieves properties from the target object.

When you run the code below, the *proxy* will intercept the `city1` property and return `Montreal, Canada`. However, when the key is not `city1`, it will use `Reflect.get()` and return the original value, so `proxy.city2` returns `Mombasa, Kenya`.

You can view a full list of available Proxy traps here.

```
const target = {
    city1: "Marseille, France",
    city2: "Mombasa, Kenya"
};

const handler = {
    get: function (target, prop, receiver) {
        if (prop === "city1") {
            return "Montreal, Canada";
        }
        return Reflect.get(...arguments);
    },
};

const proxy = new Proxy(target, handler);

console.log(proxy.city1); // Montreal, Canada
console.log(proxy.city2); // Mombasa, Kenya
```

Facade Pattern

The *Facade Pattern* is a single class that takes all of the complexity of a subsystem, and hides it. It is commonly used in JavaScript libraries and to simplify interactions with APIs. Use this pattern to create an easier interface for end users.

In the pseudo-code example below, the `VideoConverter` class provides access to the subsystem classes and is meant to direct client requests across the moving parts. The client would only interface with the `VideoConverter` class.

```
class VideoConverter {
    constructor() {}
        convertNewVideo(...args) {
        // This method can interact with `Audio`, `Video`, `Codec`, and
`Compression`
    }
}

class Audio {
    constructor() {}
    // complex subsystem code
}

class Video {
    constructor() {}
    // complex subsystem code
}

class Codec {
    constructor() {}
    // complex subsystem code
}

class Compression {
    constructor() {}
    // complex subsystem code
}
```

The Behavioral Category

Behavioral patterns streamline messages between unrelated objects in your code by delegating how objects can communicate. They encapsulate the communication behavior to decouple messages between senders and receivers.

Behavioral Patterns include:

- Iterator
- Mediator
- Observer
- Visitor

Observer Pattern

On your favorite social media platforms, other users can follow your activity and be notified when you publish new content. In the same vein, with the Observer Pattern, objects can have dependencies that "subscribe" to view changes to another object. The notifications can flow in a one-to-many relationship, i.e. one changing object can notify many other objects.

Implementation of the Observer Pattern

In the example, a basic model of the notification flow occurs between `Account` objects, which use the `.addToFollowers()` and `.removeFromFollowers()` methods to subscribe other `Account` objects to its feed activity. In the real-world, you're more likely to use the Observer pattern across different types of objects to track changes of state, but the example shows how updates can be pushed to other objects.

Here we create 3 account objects, `a`, `b`, and `c`. The `a` object uses its `.addToFollowers()` method to be 'followed' by the `b` and `c` objects. Then, the `a` object publishes a new post, "Hello, world". We can log the `b` or `c` object to see the post was added to their `feed` property.

```
class Account {
    constructor() {
        this.followers = [];
        this.feed = [];
    }
    addToFollowers(follower) {
        this.followers.push(follower);
    }
    removeFromFollowers(follower) {
        this.followers.splice(this.followers.indexOf(follower), 1);
    }
    publish(post) {
        this.followers.forEach(follower => follower.update(post));
    }
    update(post) {
      this.feed.push(post);
    }
}

let a = new Account();
let b = new Account();
let c = new Account();

a.addToFollowers(b);
a.addToFollowers(c);
```

```
o.publish("Hello, world");

console.log(c);
/* Output shows b and c objects listed in a's followers array:
{
  Account { followers: [], feed: [ 'Hello, world' ] },
  Account { followers: [], feed: [ 'Hello, world' ] }
]
*/

console.log(b);
// Account { followers: [], feed: [ 'Hello, world' ] }
```

Mediator Pattern

If you've ever picked up a rideshare app to get a ride home, you may recall that you send a request through the app, then the app "mediates" and assigns a driver to pick you up. You're not directly calling a driver to pick you up. In this

Back Next

other.

This pattern helps prevent having too many direct relationships between different classes or components and helps disparate components know about changes in application state. As a benefit, it also makes your code more reusable and easier to modify down the line since classes are not tightly coupled.

Implementation of the Mediator Pattern

In the example below, a `Passenger` object can send a request through a `RideshareApp` object, which acts as a mediator between `Driver` objects and `Passenger` objects. The interface for the `Passenger` and `Driver` objects are simpler than what you'd see in a real-world scenario and do not deal with the complexity of communicating between these two types of objects. In fact, in our example, these two types of objects don't need to know what instances exist.

```
class Passenger {
    constructor(name) {
      this.name = name;
    }
    sendRequest(rideshareapp) {
      rideshareapp.receiveRequest(this.name);
    }
}

class Driver {
    constructor(name) {
        this.name = name;
    }
    goOnline(rideshareapp) {
        rideshareapp.addDriver(this.name);
    }
}

class RideshareApp {
    constructor() {
      this.drivers = [];
      this.riders = [];
    }
    addDriver(name) {
        this.drivers.push(name);
    }
    updateDriverArray(name) {
        this.drivers.splice(this.drivers.indexOf(name), 1);
    }
    assignDriver() {
        // We will assume there are always more drivers than riders
        return
this.drivers[Math.floor(Math.random(this.drivers.length))].name;
    }
    receiveRequest(passenger) {
        const driver = this.assignDriver();
        console.log(driver);
        this.sendNotification(passenger, driver);
        this.updateDriverArray(driver);
    }
    sendNotification(passenger, driver) {
        console.log(`${driver} is assigned to ${passenger}.`)
    }
}

const rideshareapp = new RideshareApp();

const james = new Passenger("James");
const sarah = new Driver("Sarah");

rideshareapp.addDriver(sarah);
james.sendRequest(rideshareapp);
```

Putting It All Together: How to Select the Right Design Pattern

With so many patterns and anti-patterns to take into consideration, it will take time to refine your design pattern selection process. There are a few steps you can take to choose an appropriate design pattern.

- Think about the interface of each object and how it will interact with other objects. Are you encapsulating the right information in each object, or should you create a new type of object?
- Consider the specifications for each object and how you will handle each property. What other objects need awareness of this object's properties? How will you handle updates?
- Remember the high-level intent of each group of design patterns. If you are designing for object behavior versus how the object is created, review design pattern options in the Behavioral category.
- After you pick a design, review the design to see if there's any reason you should pick a different design. Is there something you need to refactor, or a problem that seems messy to handle?
- Is there anything you want to change without redesigning your code? Can you do that with your current design pattern selection? Or do you need to introduce another pattern? Remember that you can use multiple different design patterns in the same code base.

Design Patterns in Architecture

Design patterns can also be applied to system architectures. We can move "up the stack" from code units to processes and systems. Let's discuss how you could implement a couple of the design patterns you've seen applied to code, but within the context of system design.

Using the Proxy Pattern to Implement a Proxy Server

It's common to use proxy servers to streamline web traffic and protect sensitive data. As it happens, the Proxy pattern is a great fit for implementing a proxy server.

You may recall that the Proxy pattern protects access to objects by intercepting and redefining operations on the target object. That is exactly what you need for implementing a proxy server, which can be used to hide your identity from remote servers or modify requests and responses.

As we learned earlier, the pattern can increase the efficiency of requests; one of the advantages is that we can use the proxy to cache results and handle requests from multiple users.

Using the Facade Pattern to Implement Microservices

Microservices are an architectural style that structures an application as a collection of services. In theory, microservices make it easier for business units to write their own APIs that can interact with other parts of the code base.

You can use the facade pattern to implement interoperability between these isolated services. The facade is used as an interface so that the individual pieces do not become dependent or tightly coupled and the clients do not need to know about the underlying code implementation of other services.

Looking Ahead

As you continue on your coding journey, you'll encounter more design patterns and anti-patterns. In this article, we took a high-level look at anti-patterns and three categories of design patterns:

- Creational
- Behavioral
- Structural

Within each category, we walked through a couple patterns to gain familiarity with the concepts. After this article, you can start to recognize design patterns in code and practice implementing and including them in your code design process.

Free response

Given the following code example, identify what design pattern is being used and what the code does.

```
function createDog(name, breed, age) {
 return {
     name: name,
     breed: breed,
     age: age,
     getDescription() {
         return `${this.name} is a ${this.breed} that is ${this.age}
${this.age > 1 || this.age === 0 ? "years" : "year"} old.`
     }
  }
}

let moshi = createDog("Moshi", "German Shorthaired Pointer", 1);
console.log(moshi.getDescription());
```

Submit Response

Stuck? Get a Hint

Free response

Given a code example, identify what design pattern can be used to refactor the code.

```
function createAirplane(id) {
 return {
   id: id,
   nearbyAirplanes: {/* This object updates with data about nearby airplanes
based on scanADSBOut() method */ },
     getLocation() {
       // code to get latitude, longitude, and altitude
       return [lat, lon, alt]
     },
     scanADSBOut() {
       // Code that uses an antenna from nearby airplanes to find nearby
airplanes based on proximity of latitude, longitude, and altitude
     },
     checkForCrash() {
       let location = this.getLocation();
       // Code that checks if locations of nearby airplanes are within X miles
of each other and calculates trajectory based on location and altitude
```

```
  },

 }
}

let flight1 = createAirplane(1);
let flight2 = createAirplane(2);
let flight3 = createAirplane(3);

flight1.scanADSBOut(); // This method scans for any planes that are nearby and
stores the values in the `nearbyAirplanes` property
console.log(flight1) // After logging the `flight1` object, it'll display the
latitude, longitude, and altitude in the `nearbyAirplanes` object, along with
other properties
flight1.checkForCrash(); // This does a brute force check to check for an
impending crash by using the values in the `nearbyAirplanes` property and the
airplane's current location, using the `getLocation()` method
```

Submit Response

Stuck? Get a Hint

Back Next

Back Next